GENERACIÓN AUTOMÁTICA DE PUERTOS EN ERLANG

FERNANDO SUÁREZ JIMÉNEZ

MÁSTER EN INGENIERÍA EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería Informática

Convocatoria: 3 de septiembre de 2018

Calificación: 9

Director:

Manuel Montenegro Montes

Autorización de difusión

Fernando Suárez Jiménez

Madrid, 2 de septiembre de 2018

El/la abajo firmante, matriculado/a en el Máster en Ingeniería en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "GENERACIÓN AUTOMÁTICA DE PUERTOS EN ERLANG", realizado durante el curso académico 2017-2018 bajo la dirección de Manuel Montenegro Montes en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

Son varios los mecanismos de interoperabilidad ofrecidos por el lenguaje Erlang para realizar conexiones con código C. En este trabajo se han estudiado varios de estos mecanismos con el fin de encontrar el más adecuado para la generación de *bindings* que permitan el uso de librerías de C en programas Erlang. La primera parte de este trabajo se ha centrado en la realización de varias pruebas de concepto con la librería SDL de C evaluando el uso de varios de estos mecanismos, como las NIFs o los puertos. Una vez realizadas estas pruebas y determinado el método más adecuado, se ha desarrollado en Erlang una herramienta automática de generación de *bindings* que se encarga de producir el código C y el código Erlang necesario para poder utilizar estas librerías de C en programas Erlang de una manera sencilla y completamente funcional. Esta herramienta requiere de un fichero de especificación que establece la correspondencia entre los elementos de ambos lenguajes y ha de ser escrito manualmente por el programador. Entre las características del código generado por esta herramienta podemos destacar: manejo de tipos básicos de C, funciones, punteros, *arrays*, estructuras, uniones y enumerados, tratamiento de macros, soporte para funciones de orden superior y gestión automatizada de memoria dinámica para punteros de C. Todas estas características permiten la creación de programas Erlang plenamente funcionales que hacen uso de librerías de C, como es el caso del programa de prueba realizado para la librería SDL.

Palabras clave

Binding, C, Comunicación Erlang-C, Erlang, Generación automática de código, Interoperabilidad, Librería, NIF, Puerto, SDL

El código desarrollado en este proyecto puede encontrarse en los siguientes repositorios:

- Prueba de concepto de SDL en C:
  https://github.com/fersj/TFM-TestSDLGame
- Herramienta de generación de código:
  https://github.com/fersj/ErlangSDL

Abstract

There are several interoperability mechanisms offered by the Erlang language to make connections with C code. In this work we have studied several of these mechanisms in order to find the most suitable for the generation of bindings that allow the use of C libraries in Erlang programs. The first part of this work focuses on carrying out several proof-of-concept tests with the SDL C library, evaluating the use of several of these mechanisms, such as NIFs or ports. Once these tests have been carried out and the most appropriate method has been determined, an automatic generation tool for bindings has been developed in Erlang, which is responsible for producing the C and Erlang code necessary to use these C libraries in Erlang programs in a simple and fully functional way. This tool requires a specification file that establishes the correspondence between the elements of both languages and has to be written manually by the programmer. Among the features of the code generated by this tool we can highlight: C basic type management, functions, pointers, arrays, structures, unions and enumerations, macro management, higher-order function support and automatic dynamic memory management for C pointers. All these features allow the creation of fully functional Erlang programs that use C libraries, as is the case with the test program for the SDL library.

Keywords

Automatic code generation, Binding, C, Erlang, Erlang-C communication, Interoperability, Library, NIF, Port, SDL

The code developed in this project can be found in the following repositories:

- Proof-of-concept test of SDL in C:
  https://github.com/fersj/TFM-TestSDLGame
- Code generation tool:
  https://github.com/fersj/ErlangSDL

Índice general

Agradecimientos

Gracias a mi tutor, Manuel Montenegro Montes, por guiarme a lo largo de estos meses en el desarrollo de este trabajo, por sus consejos, su tiempo y su dedicación. Sin sus ideas y aportaciones este proyecto no habría sido posible.

Capítulo 1

Introducción

Por todos es sabido que el lenguaje C es uno de los más extendidos del mundo (ver figura 1.1), no solo por su antigüedad sino también por su flexibilidad, ya que la gran cantidad de librerías de la que dispone para casi cualquier propósito lo hace una opción a tener en cuenta a la hora de desarrollar una gran variedad de sistemas. No es así para el caso de Erlang, un lenguaje funcional mucho menos extendido que, a pesar de contar con cierta cantidad librerías y *frameworks* para multitud de usos, puede presentar algunas carencias en según qué contextos de uso.

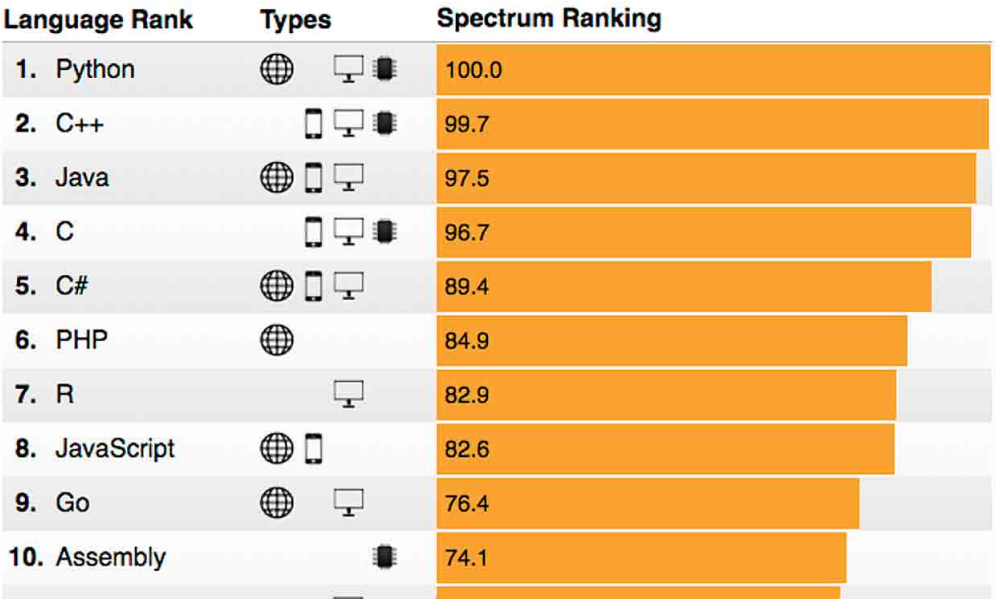

Figura 1.1: *Lenguajes de programación más populares en 2018, según IEEE Spectrum [6]*

Para dar solución a este problema, Erlang ofrece diversos mecanismos que permiten la interoperabilidad entre estos dos lenguajes, de modo que, haciendo uso de estos mecanismos, se pueden crear *bindings* de librerías de C para que puedan ser usadas en Erlang.

La motivación principal de este trabajo es desarrollar una herramienta que permita generar de manera automática los *bindings* necesarios para hacer uso de librerías de C en Erlang mediante alguno de los mecanismos de interoperabilidad ofrecidos por este lenguaje. Para ello es necesario que el programador especifique previamente la correspondencia entre las estructuras de datos y funciones de C y sus contrapartes en Erlang. Como caso particular, se ha pensado en la librería SDL[1] (*Simple DirectMedia Layer*) como punto de partida para la generación de los *bindings* con esta herramienta. Se trata de una librería para el desarrollo de videojuegos 2D en C/C++, aunque cuenta con *bindings* para otros lenguajes. El motivo de elegir esta librería no es otro que la falta de herramientas de desarrollo de videojuegos en un lenguaje como Erlang.

1.1. Estado del arte

En relación con el tema de este trabajo, existen en la actualidad algunos proyectos que ha seguido un enfoque similar al que aquí se plantea. Uno de estos proyectos es ***Nifty*** [7, 10], cuya premisa es la generación de *bindings* de librerías C para Erlang mediante el uso de funciones implementadas de manera nativa (NIFs, ver sección 2.2). Esta herramienta se encarga de generar las NIFs de la librería de C a partir de su fichero fuente (`.c`) y de cabecera (`.h`). No obstante esta herramienta presenta una serie de limitaciones, de entre las cuales podemos destacar las siguientes:

- Los punteros a funciones solo son compatibles parcialmente.
- No ofrece soporte para *struct* y *unions* anónimos.
- No se generan NIFs para funciones con tipos no soportados.

[1] https://www.libsdl.org

- No soporta funciones con número variable de argumentos.

Este proyecto está siendo llevado a cabo por PARAPLUU, un grupo de investigación en ciencias de la computación de la Universidad de Uppsala (Suecia) y se encuentra aún en desarrollo.

Siguiendo con soluciones que permiten ejecutar en Erlang código de otros lenguajes encontramos ***Pytherl*** [13], un proyecto desarrollado por Michal Ptaszek que permite la ejecución de código Python en Erlang. Consiste en un módulo de Erlang (*pytherl*) que proporciona mecanismos para la ejecución de funciones escritas en Python, especificando el módulo, el nombre de la función a ejecutar y sus argumentos.

A continuación se muestra un ejemplo de uso de este módulo tomado de la web del autor [13],

```
pytherl:call("re", "re.sub", ["Erlang", "Python", "Hello from Erlang!"]).
```

que es equivalente a ejecutar el siguiente código en Python:

```
import re

re.sub("Erlang", "Python", "Hello from Erlang!")
```

Si nos centramos en la generación de *bindings* de librerías de C, encontramos proyectos como ***c-for-go*** [9], una herramienta desarrollada por Sphere Software con licencia MIT que se encarga de generar *bindings* de librerías de C/C++ para el lenguaje de programación Go. La herramienta crea automáticamente los *bindings* a partir de las cabeceras de C (`.h`) y un fichero de manifiesto YAML. En la figura 1.2 puede verse un esquema de funcionamiento de esta herramienta proporcionado por sus creadores.

Existen también otro tipo de herramientas que son capaces de conectar código C/C++ con una amplia variedad de lenguajes de alto nivel. Es el caso de ***SWIG*** [4]. Esta herramienta de código abierto escrita en C/C++, es capaz de conectar programas escritos en C/C++ con diferentes lenguajes de programación como Javascript, Perl, PHP, Python, Ruby, D, Go o Java. Para realizar la conexión entre los lenguajes se hace uso de un fichero *interface* (`.i`)

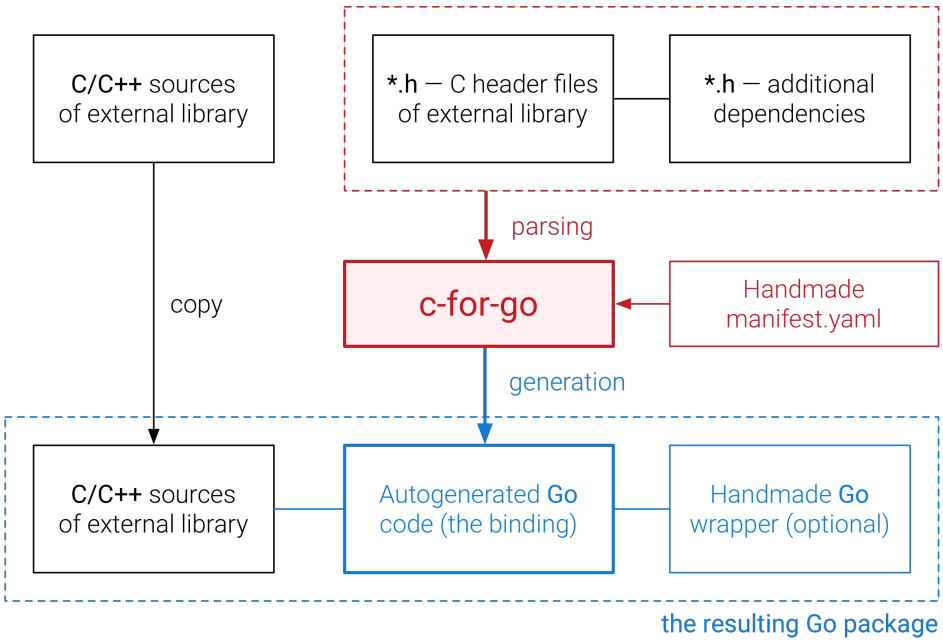

Figura 1.2: *Esquema de generación de bindings de la herramienta c-for-go (imagen tomada de la web del autor [9])*

donde el programador especifica las distintas variables y funciones C/C++ que se quieren ejecutar desde el lenguaje en cuestión.

A continuación puede verse un ejemplo de código C y su correspondiente fichero *interface* tomados de la propia web del proyecto (http://www.swig.org/tutorial.html):

```
/* File : example.c */

#include <time.h>
double My_variable = 3.0;

int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time()
```

```
{
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

```
/* example.i */
%module example
%{
/* Put header files here or function declarations like below */
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

El siguiente listado de comandos (también tomado de la propia web del proyecto) muestra un ejemplo de uso de esta herramienta para Java, haciendo uso de los ficheros anteriores.

```
$ swig -java example.i
$ gcc -c example.c example_wrap.c -I/c/jdk1.3.1/include -I/c/jdk1.3.1/include
   /win32
$ gcc -shared example.o  example_wrap.o -mno-cygwin -Wl,--add-stdcall-alias
   -o example.dll
$ cat main.java
 public class main {
   public static void main(String argv[]) {
     System.loadLibrary("example");
     System.out.println(example.getMy_variable());
     System.out.println(example.fact(5));
     System.out.println(example.get_time());
   }
 }
$ javac main.java
$ java main
3.0
120
Mon Mar  4 18:20:31  2002
```

También se pueden encontrar algunos proyectos que tratan de portar la librería SDL a Erlang. Un ejemplo de ellos es el proyecto ***esdl2*** [12] desarrollado por Nine Nines. Este proyecto trata de implementar la librería SDL, así como algunas de sus extensiones (*SDL_image* y *SDL_ttf*), haciendo uso de NIFs como mecanismo de comunicación entre Erlang y C.

Como último ejemplo de esto tenemos el proyecto ***esdl*** [14] de Aur Saraf. Este proyecto se acerca más a la herramienta realizada en este proyecto, ya que implementa un generador de código desarrollado en Erlang que se encarga de generar los *bindings* para esta librería. En este caso el método de comunicación entre Erlang y C utilizado son los *port drivers* (ver sección 2.4). Tanto este proyecto como el anterior se encuentran inacabados y, al menos el primero de ellos, aún en desarrollo.

1.2. Objetivos

Los objetivos de este proyecto se resumen, principalmente, en dos puntos:

1. Desarrollo de un sistema que permita la generación automática de código de interoperabilidad para el uso de librerías C en programas Erlang. Este sistema debe generar una serie de *bindings* a partir de unas especificaciones dadas por el programador. Además, a la hora de generar el código se deben tener en cuenta los siguientes aspectos:

   - Generación de mecanismos para el manejo de punteros, *arrays* y otros tipos de datos como estructuras, uniones o enumerados.
   - Inclusión de un sistema de gestión automática de memoria dinámica para punteros C desde Erlang.
   - Compatibilidad con funciones de orden superior.

2. Aplicación de la herramienta desarrollada a la librería SDL con el fin de poder realizar un prototipo sencillo funcional desarrollado completamente en Erlang haciendo uso de los *bindings* generados. Los requisitos para este prototipo son:

- Manejo de elementos gráficos 2D.
- Carga de ficheros de imágenes.
- Tratamiento de eventos de teclado.

1.3. Plan de trabajo

Para el desarrollo de este trabajo se han establecido dos grandes fases bien diferenciadas. A continuación se describen cada una de ellas:

1. **Creación de pruebas de concepto con la librería SDL** (capítulo 2), con el fin de tener una primera toma de contacto con la librería, valorar algunos de los mecanismos de interoperabilidad ofrecidos por Erlang y seleccionar el que sea más adecuado para las necesidades del proyecto. Esta fase se divide a vez en las siguientes tareas:

   a) Desarrollo de una prueba de concepto con SDL en código C (sección 2.7).

   b) Desarrollo de una prueba de concepto con SDL en Erlang mediante el uso de NIFs como método de interoperabilidad (sección 2.8).

   c) Desarrollo de una prueba de concepto con SDL en Erlang mediante el uso de puertos como método de interoperabilidad (sección 2.9).

2. **Creación del sistema de generación de *bindings* para el uso de librerías C en Erlang**. El desarrollo de este sistema consta de las siguientes etapas:

   a) Generación de mecanismos para el manejo de tipos básicos, macros y funciones de la librería con el fin de poder desarrollar en Erlang un prototipo inicial muy básico haciendo uso del código generado (capítulo 3).

   b) Generación de mecanismos para el manejo de enumerados, estructuras y uniones con el fin de incrementar la funcionalidad del prototipo de SDL en Erlang (capítulo 4).

c) Generación de mecanismos para el manejo de punteros C desde código Erlang (capítulo 5). Hacer uso del código generado hasta el momento para incrementar el prototipo y equipararlo en funcionalidades al desarrollado en en la fase 1.

d) Generación de mecanismos para el manejo de *arrays* (capítulo 6).

e) Inclusión de un sistema para gestionar de forma automática la memoria dinámica de los punteros C utilizados desde código Erlang (capítulo 7).

f) Generación de *bindings* para funciones de orden superior (capítulo 8).

Una vez introducido el contexto de este proyecto, los objetivos a conseguir y el plan de trabajo, se va a proceder en los siguientes capítulos a desarrollar cada una de las etapas aquí descritas.

Chapter 1

Introduction

Everyone knows that the C language is one of the most widespread in the world (see figure 1.1), not only for its age but also for its flexibility, since the large number of libraries available for almost any purpose makes C be an option to take into consideration when developing any system. It is not so for the case of Erlang, a much less extended functional language that, despite having a certain number of libraries and *frameworks* for a multitude of uses, may present some deficiencies depending on the use context.

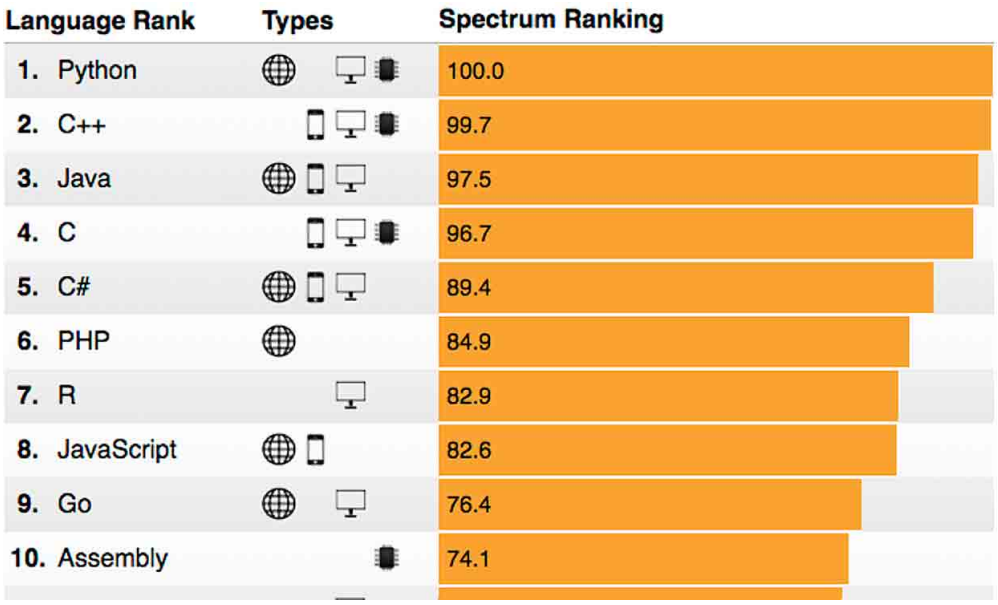

Figure 1.1: *Top 10 programming languages of 2018, according to IEEE Spectrum [6]*

To solve this problem, Erlang offers several mechanisms that allow interoperability be-

tween these two languages, so that, by using these mechanisms, one can create bindings of C libraries so they can be used in Erlang.

The main motivation of this work is to develop a tool that allows one to automatically generate the bindings necessary to use C libraries in Erlang by means of some of the interoperability mechanisms offered by this language. Specifically, the SDL library[1] (*Simple DirectMedia Layer*) has been thought of as the starting point for the generation of bindings with this tool. It is a library for 2D video game development in C/C++, although it has bindings for other languages. The reason for choosing this library is the lack of videogame development tools in a language like Erlang.

1.1. State of the art

In relation to the subject of this work, there are currently some projects that have followed an approach similar to the one proposed here. One of these projects is ***Nifty*** [7, 10], whose premise is the generation of bindings from C libraries for Erlang through the use of NIFs (see section 2.2). This tool is responsible for generating the NIFs of the C library from its source file (`.c`) and header (`.h`). However, this tool has a number of limitations, among which we can highlight the following:

- Function pointers are only partially compatible.
- It does not offer support for anonymous *struct* and *unions*.
- NIFs are not generated for functions with unsupported types.
- It does not support functions with a variable number of arguments.

This project is being carried out by PARAPLUU, a research group in computer science of the University of Uppsala (Sweden) and is still in development.

[1] https://www.libsdl.org

Following with solutions that allow one to execute in Erlang code from other languages, we find ***Pytherl*** [13], a project developed by Michal Ptaszek that supports the execution of Python code in Erlang. It consists of an Erlang module (*pytherl*) that provides mechanisms for the execution of functions written in Python, by specifying the module, the name of the function to execute and its arguments.

Below is a use example of this module taken from the author's website [13],

```
pytherl:call("re", "re.sub", ["Erlang", "Python", "Hello from Erlang!"]).
```

which is equivalent to executing the following code in Python:

```
import re

re.sub("Erlang", "Python", "Hello from Erlang!")
```

If we focus on the generation of bindings from C libraries, we find projects like ***c-for-go*** [9], a tool developed by Sphere Software with MIT license that is responsible for generating bindings from C/C++ libraries for the Go programming language. The tool automatically creates the bindings from the C headers (`.h`) and a manifest YAML file. Figure 1.2 depicts an operation scheme of this tool provided by its creators.

There are also other types of tools that are able to connect C/C++ code with a wide variety of high level languages. This is the case of ***SWIG*** [4]. This open source tool written in C/C++, is able to connect programs written in C/C++ with different programming languages such as Javascript, Perl, PHP, Python, Ruby, D, Go or Java. To make the connection between the languages we use a file *interface* (`.i`) where the programmer specifies the different variables and C/C++ functions that we want to execute from the language in question.

Below is an example of C code and its corresponding *interface* file taken from the project's website (http://www.swig.org/tutorial.html):

```
/* File : example.c */

#include <time.h>
double My_variable = 3.0;
```

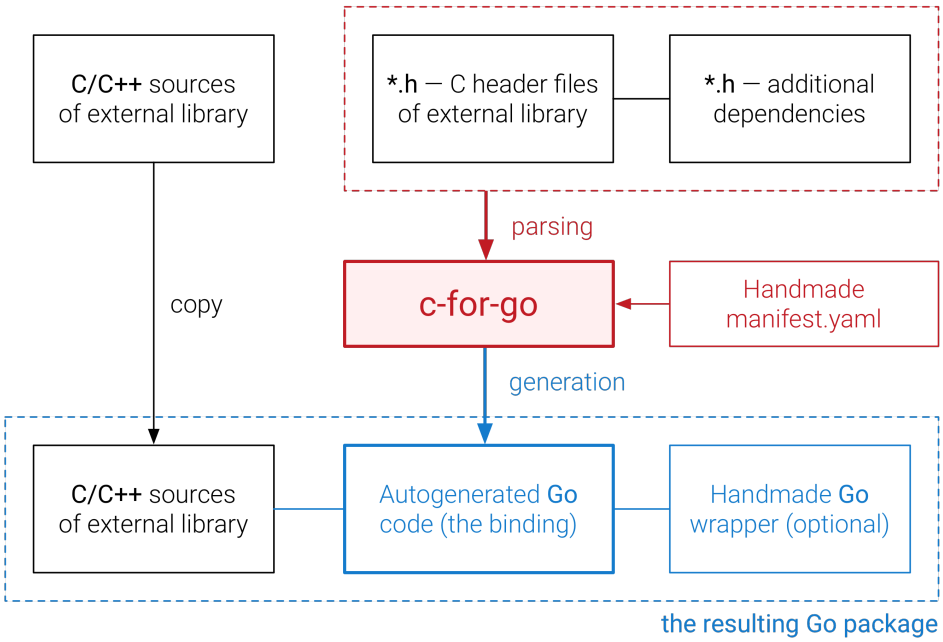

Figure 1.2: *Scheme of generation of bindings of the tool c-for-go (image taken from the author's website [9])*

```
int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time()
{
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

```
/* example.i */
%module example
%{
/* Put header files here or function declarations like below */
```

```
 extern double My_variable;
 extern int fact(int n);
 extern int my_mod(int x, int y);
 extern char *get_time();
 %}

 extern double My_variable;
 extern int fact(int n);
 extern int my_mod(int x, int y);
 extern char *get_time();
```

The following list of commands (also taken from the project's website) shows an example of using this tool for Java, making use of the previous files.

```
$ swig -java example.i
$ gcc -c example.c example_wrap.c -I/c/jdk1.3.1/include -I/c/jdk1.3.1/include
   /win32
$ gcc -shared example.o  example_wrap.o -mno-cygwin -Wl,--add-stdcall-alias
   -o example.dll
$ cat main.java
 public class main {
   public static void main(String argv[]) {
     System.loadLibrary("example");
     System.out.println(example.getMy_variable());
     System.out.println(example.fact(5));
     System.out.println(example.get_time());
   }
 }
$ javac main.java
$ java main
3.0
120
Mon Mar  4 18:20:31  2002
```

You can also find some projects that try to port the SDL library to Erlang. An example of them is the project ***esdl2*** [12] developed by Nine Nines. This project tries to implement the SDL library, as well as some of its extensions (*SDL_image* and *SDL_ttf*), making use of NIFs as communication mechanism between Erlang and C.

As a last example of this we have the project ***esdl*** [14] of Aur Saraf. This project is closer to the tool made in this project, since it implements a code generator developed in Erlang

that is responsible for generating bindings for this library. In this case, the communication method between Erlang and C used are the *port drivers* (see section 2.4). Both this project and the previous one are unfinished and, at least the first of them, still in development.

1.2. Goals

The goals of this project are summarized, mainly, in two points:

1. Development of a system for the automatic generation of interoperability code for the use of C libraries in Erlang programs. This system must generate a series of bindings based on the specifications given by the programmer. In addition, when generating the code, the following aspects should be taken into account:
   - Generation of mechanisms for handling pointers, arrays and other data types such as structures, unions or enumeration.
   - Inclusion of an automatic dynamic memory management system for C pointers from Erlang.
   - Compatibility with higher order functions.
2. Application of the developed tool to the SDL library in order to create a simple functional prototype developed completely in Erlang from the generated bindings. The requirements for this prototype are:
   - Management of 2D graphic elements.
   - Loading image files.
   - Handling keyboard events.

1.3. Workplan

Two well defined phases have been established for the development of this work. Each of them is described in the following:

1. **Creation of proof-of-concept tests with the SDL library** (chapter 2), in order to have a first contact with the library, we evaluate some of the interoperability mechanisms offered by Erlang and select which is most suitable for the needs of the project. This phase is divided into the following tasks:

   a) Development of a proof-of-concept test with SDL in C code (section 2.7).

   b) Development of a proof-of-concept test with SDL in Erlang through the use of NIFs as an interoperability method (section 2.8).

   c) Development of a proof-of-concept test with SDL in Erlang through the use of ports as an interoperability method (section 2.9).

2. **Creation of the bindings generation system for the use of C libraries in Erlang**. The development of this system consists of the following stages:

   a) Generation of mechanisms for handling basic types, macros and functions of the library in order to develop in Erlang a very basic initial prototype using the generated code (chapter 3).

   b) Generation of mechanisms for handling enumerations, structures and unions in order to increase the functionality of the SDL prototype in Erlang (chapter 4).

   c) Generation of mechanisms for handling C pointers from Erlang code (chapter 5). Application of the code generated so far to increase the prototype and compare it in functionalities to the one developed in phase 1.

   d) Generation of mechanisms for the management of arrays (chapter 6).

   e) Inclusion of a system to automatically manage the dynamic memory of the C pointers used from Erlang code (chapter 7).

   f) Generation of bindings for higher order functions (chapter 8).

Once the context of this project, the goals to be achieved and the work plan have been introduced, the following chapters will elaborate on each of the stages described here.

Capítulo 2

Preliminares. Mecanismos de interoperabilidad entre Erlang y C

En este capítulo se van a describir los diferentes mecanismos de interoperabilidad entre Erlang y C que existen en la actualidad. Se comenzará dando una visión general del lenguaje Erlang, enumerando sus principales características. Después se detallarán las distintas formas en las que puede ejecutarse código escrito en C desde Erlang y se estudiará la eficiencia de cada una de ellas. Finalmente, se explorarán dos de estos mecanismos con el fin de examinar su adecuación a uno de los objetivos de este trabajo, que es el uso de la librería gráfica SDL desde Erlang. Para esto último se partirá de un programa sencillo, a modo de prueba de concepto.

2.1. Características básicas de Erlang

Antes de comenzar con los diferentes mecanismos de interoperabilidad es conveniente exponer de forma breve las diferentes características que lenguaje Erlang como punto de partida.

Erlang es un lenguaje de programación funcional desarrollado por Ericsson y Ellemtel Computer Science Laboratories pensado para el desarrollo de sistemas concurrentes y distribuidos. Entre sus características cabe destacar las siguientes [1, 2]:

- Permite la creación de procesos propios que son gestionados directamente por la má-

quina virtual de Erlang, de modo independiente a la gestión de procesos realizada por el sistema operativo. Son procesos ligeros, lo que hace que Erlang pueda trabajar con enormes cantidades de dichos procesos de forma eficiente.

- Ofrece primitivas para el manejo de concurrencia mediante paso de mensajes. Estos mensajes son la única forma de comunicación entre los procesos Erlang y son enviados de forma asíncrona. Gracias a este esquema de comunicación los programas concurrentes en Erlang son fácilmente implementables como sistemas distribuidos.

- Para trabajar con sistemas distribuidos, ofrece la posibilidad de crear nodos que pueden ser ejecutados en la misma o en diferentes máquinas. Estos nodos pueden ser creados como "ocultos", lo cual requiere realizar la conexión con otros nodos de forma explicita, evitando así que se creen conexiones indeseadas.

- Permite estructurar programas de gran tamaño en unidades mas manejables llamadas módulos. Estos módulos contienen, principalmente, funciones.

- Como lenguaje funcional, las variables en Erlang tienen asignación simple, es decir, una vez se les asigna un valor éste ya no puede cambiar. Además, cuenta con tipado dinámico, ya que la comprobación de tipos se realiza en tiempo de ejecución.

- Presenta una gran tolerancia a fallos. Para ello proporciona distintas construcciones (por ejemplo, árboles de supervisión) que permiten el reinicio de procesos en caso de fallo.

- Permite realizar cambios en el código en sistemas en ejecución (*Hot Code Swapping*).

2.2. NIFs

Las NIFs (*Native Implemented Functions*) de Erlang son el mecanismo de interoperabilidad más básico y eficiente que permite llamar a código C. Su uso está recomendado para

funciones síncronas que realizan una serie de cálculos (más o menos simples) y devuelven un resultado.

Una NIF es una función implementada en C que puede ser llamada desde Erlang como cualquier otra función nativa que pertenece a un módulo. El código C es compilado como una librería dinámica y ésta se carga mediante el código Erlang en tiempo de ejecución [1].

A continuación se presenta un ejemplo simple de función NIF para realizar el cálculo de un número de la sucesión de Fibonacci. Consta de 3 ficheros:

- *fibonacci.c*: código C con la función de Fibonacci.

```c
#include <stdio.h>

int fibonacci(int n) {
   int first = 0, second = 1, next, c;

   printf("First %d terms of Fibonacci series are :-\n", n);

   for (c = 0; c < n; c++) {
      if (c <= 1)
         next = c;
      else {
         next = first + second;
         first = second;
         second = next;
      }
      printf("%d\n", next);
   }

   return 0;
}
```

- *erlang_fib.c*: código C que contiene las NIFs que serán llamadas desde Erlang. Para establecer la comunicación y la correspondencia de datos entre Erlang y C se hace uso de la librería *erl_nif.h*.

```c
#include <erl_nif.h>

extern int fibonacci(int n);
```

```c
static ERL_NIF_TERM fibonacci_nif (ErlNifEnv *env, int argc,
                                   const ERL_NIF_TERM argv[]) {
   int n, result;
   if (!enif_get_int(env, argv[0], &n)) {
      return enif_make_badarg(env);
   }

   result = fibonacci(n);
   return enif_make_int(env, result);
}

// Especifica la correspondencia entre las funciones de Erlang
// y las de C. En este caso, indica que la función de Erlang
// fibonacci/1 se corresponde con la función fibonacci_nif de C.

static ErlNifFunc funcs[] = {
    {"fibonacci", 1, fibonacci_nif}
};

ERL_NIF_INIT(fibonacci, funcs, NULL, NULL, NULL, NULL)
```

- *fibonacci.erl*: código Erlang que se encarga de cargar las NIFs definidas en *erlang_fib.c*.

```erlang
-module(fibonacci).
-export([fibonacci/1]).

% Al cargar el módulo Erlang fibonacci, se llamará a la función
% init del mismo:
-on_load(init/0).

% La función init carga la librería dinámica escrita en C.
init() ->
   ok = erlang:load_nif("./erlang_fib", 0).

fibonacci(_N) ->
   exit(nif_library_not_loaded).
```

Una vez compilado el fichero *erlang_fib.c* como librería dinámica y generado el fichero *erlang_fib.so* (en el caso de sistemas UNIX) se compila el fichero *fibonacci.erl* y se carga

como un módulo Erlang corriente. Este último realiza la carga de la librería *erlang_fib.so* asociando las NIFs que contiene con las funciones del módulo Erlang, de modo que al llamar a la función `fibonacci` del módulo Erlang se ejecuta la función de la librería de C. Si la carga del módulo C no se realizó con éxito, al llamar a la función `fibonacci` de Erlang se producirá un error mediante la llamada `exit(nif_library_not_loaded)`.

La principal ventaja de este mecanismo es su rapidez a la hora de llamar código C, debido a que la librería de NIFs se vincula de forma dinámica al proceso Erlang, por lo que no requiere de ningún cambio de contexto. No obstante, eso a su vez supone una desventaja, ya que si se produce un error en una NIF, no sólo se verá afectado el proceso Erlang que la está llamando, sino que también se producirá una parada de la máquina virtual de Erlang [1]. Además, existe otro inconveniente relacionado con el tiempo de ejecución de las NIFs, ya que éstas deben ejecutarse y devolver el resultado en un corto periodo de tiempo (menor a 1ms) para evitar interferir con el planificador de procesos de Erlang.

2.3. Ports

Este mecanismo permite la interoperabilidad entre Erlang y C mediante la creación de un puerto que es el que se encarga de realizar todas las tareas de comunicación entre el programa en Erlang y el programa externo en C. El proceso Erlang que crea el puerto (*port owner*) es el único que puede encargarse de enviar y recibir información a través del mismo, de modo que si este proceso termina, el puerto también se cierra [1].

La comunicación entre Erlang y C a través de puertos se realiza mediante secuencias de bytes. Cada vez que el *port owner* envía una lista de bytes al puerto, dicha lista es recibida por el proceso C a través de su entrada estándar. Cuando el programa C escribe a través de la salida estándar, la información escrita se traduce a una secuencia de bytes, que es recibida por el *port owner* como un mensaje.

Los mensajes que se envían desde Erlang al puerto son precedidos de forma automática por su longitud. Mientras que, en el caso de C, esta longitud debe ser incluida en el

mensaje de forma explícita por parte del programador. La figura 2.1 muestra el esquema de comunicación entre ambos programas [1].

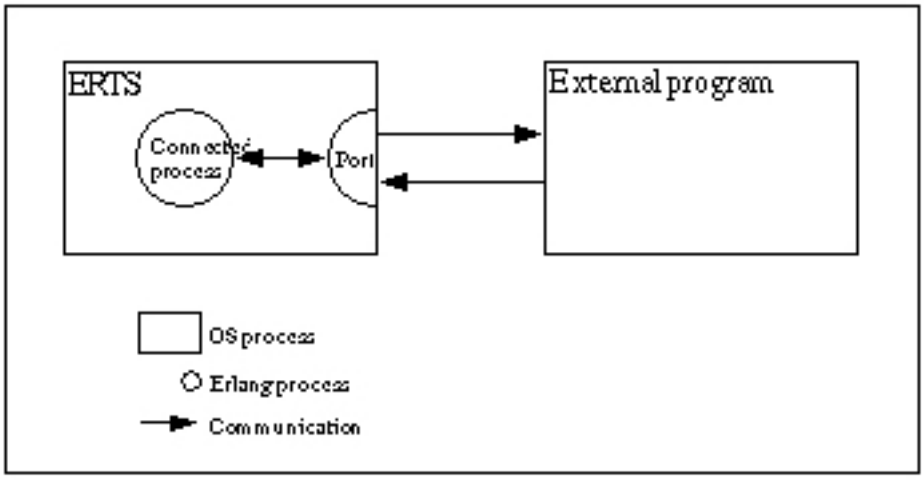

Figura 2.1: *Esquema de comunicación de puertos Erlang [1]*

El siguiente ejemplo de uso de puertos (obtenido de la documentación oficial de Erlang [1]) consta de los siguientes elementos:

- *complex1.erl*: Código Erlang que se encarga de la creación del puerto en un proceso aparte, la serialización de los datos mediante listas de bytes, su envío a través de dicho puerto, y la recepción del resultado de la función solicitada (`foo` o `bar`) que será ejecutada por el código C. El mensaje enviado al puerto contiene un identificador de la función a realizar (1 en el caso de `foo`, 2 en el caso de `bar`), seguido del argumento necesario para su ejecución.

```
-module(complex1).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

% Inicia el port owner como proceso aparte
start(ExtPrg) ->
   spawn(?MODULE, init, [ExtPrg]).

% Envía un mensaje al port owner para que finalice
```

```
stop() ->
   complex ! stop.

% Envían solicitudes de llamada al port owner
foo(X) ->
   call_port({foo, X}).
bar(Y) ->
   call_port({bar, Y}).

% Envía un mensaje al port owner y espera una respuesta
call_port(Msg) ->
   complex ! {call, self(), Msg},
   receive
      {complex, Result} ->
         Result
   end.

% Inicialización del port owner.
init(ExtPrg) ->
   register(complex, self()),
   process_flag(trap_exit, true),
   % La función open_port se encarga de arrancar el proceso
   % ExtPrg y de crear una tubería (pipe) de comunicación entre
   % el puerto y dicho proceso. La opción {packet, 2} indica que
   % cada mensaje irá precedido por dos bytes que indican la
   % longitud del mismo.
   Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
   loop(Port).

% Bucle principal del port owner
loop(Port) ->
   receive
      % Cuando recibe una petición de llamada por parte
      % de un proceso, traduce (mediante ‘encode‘) la petición
      % a una secuencia de bytes, y envía la información al
      % puerto. Después espera a recibir la respuesta enviada
      % por el proceso en C, y traduce la secuencia de bytes
      % recibida en un término de Erlang (mediante ‘decode‘)
      {call, Caller, Msg} ->
         Port ! {self(), {command, encode(Msg)}},
         receive
            {Port, {data, Data}} ->
               Caller ! {complex, decode(Data)}
```

```
            end,
            loop(Port);

        % Cuando recibe una petición de cierre, redirige esa
        % petición al puerto, y espera a que éste le envíe el
        % mensaje de confirmación de parada.
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;

        % Si el proceso C aborta, el port owner también
        % lo hará
        {'EXIT', Port, Reason} ->
            exit(port_terminated)
    end.

encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

- *port.c*: Este código C se encarga de recibir, a través de la entrada estándar, el mensaje enviado desde Erlang al puerto, ejecutar la función solicitada (`foo` o `bar`) y enviar el resultado a través del puerto. Para ello hace uso de las funciones `read_cmd` y `write_cmd` (implementación omitida por simplicidad, puede consultarse en la documentación de Erlang [1]) que se encargan de leer la secuencia de bytes con el identificador de la función a realizar y el argumento de la misma y escribir en el puerto el resultado de la función ejecutada, también como secuencia de bytes.

```
typedef unsigned char byte;

int main() {
    int fn, arg, res;
    byte buf[100];
```

```
    // Cuando se envía la señal de cierre desde el
    // port_owner, se envía el carácter EOF a la
    // salida estándar del proceso C. En este caso
    // la función read_cmd devuelve 0.
    while (read_cmd(buf) > 0) {
        // La secuencia recibida tiene dos bytes:
        // el primero es el código de operación y
        // el segundo es el argumento recibido
        fn = buf[0];
        arg = buf[1];

        if (fn == 1) {
            res = foo(arg);
        } else if (fn == 2) {
            res = bar(arg);
        }

        // Escribir el resultado en el buffer
        buf[0] = res;
        write_cmd(buf, 1);
    }
}
```

Este mecanismo de comunicación es válido tanto para C como para otros lenguajes. Su principal ventaja reside en que, al ser ejecutado el código C como un programa independiente, cualquier fallo que ocurra durante la ejecución de éste no causará la caída de la máquina virtual de Erlang y, por tanto, el programa Erlang podrá seguir ejecutándose. Por otra parte, esto da lugar a que la comunicación sea menos eficiente que con otros métodos como las *NIFs*.

2.4. Port Drivers

Los *port drivers* de Erlang funcionan de manera similar a los puertos presentados en la sección 2.3 con la diferencia de que en este caso el código C no se ejecuta como un programa independiente, sino que se carga como una librería dinámica con estructura de controlador

(*port driver*) [1], tal y como puede verse representado en la Fig. 2.2.

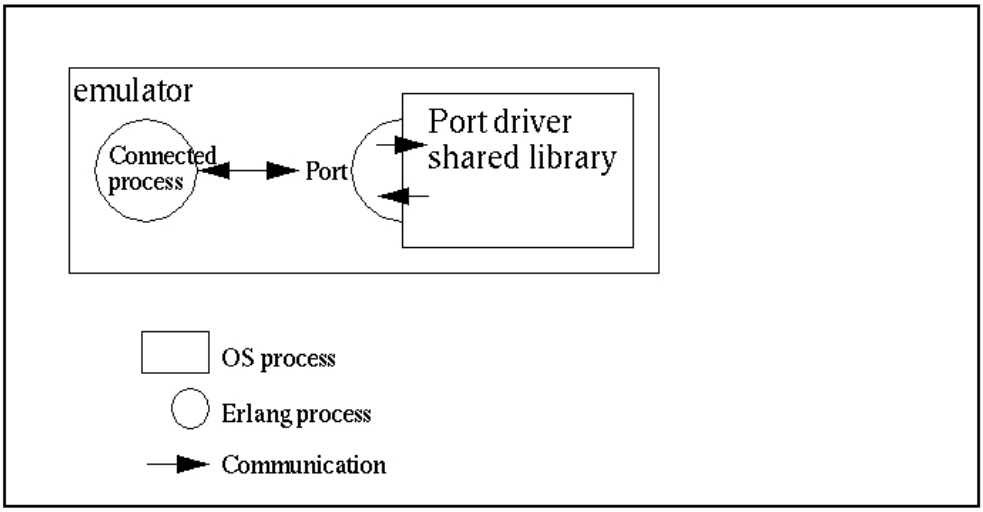

Figura 2.2: *Esquema de comunicación de los port drivers [1]*

El proceso Erlang crea el puerto, que en este caso va asociado al *port driver*, y realizada la comunicación a través del mismo. A diferencia del ejemplo anterior, el controlador debe estar cargado antes de inicializar el puerto [1]. En el siguiente listado se muestra el proceso de carga del driver y de inicialización del puerto.

```
start(SharedLib) ->
   case erl_ddll:load_driver(".", SharedLib) of
      ok -> ok;
      {error, already_loaded} -> ok;
      _ -> exit({error, could_not_load_driver})
   end,
   spawn(?MODULE, init, [SharedLib]).

init(SharedLib) ->
   register(complex, self()),
   Port = open_port({spawn, SharedLib}, []),
   loop(Port).
```

El código C, en este caso, se comunica con el puerto haciendo uso de las funciones, estructuras y constantes proporcionadas por el fichero de cabecera *erl_ driver.h*, recibiendo

los datos enviados desde Erlang y proporcionando los resultados de las operaciones realizadas como se hacía en el ejemplo de la sección 2.3.

Este mecanismo, al igual que las NIFs, es muy eficiente debido a que el *port driver* se vincula dinámicamente al proceso de Erlang, por lo que las llamadas no requieren cambios de contexto entre el proceso Erlang y el proceso C. Sin embargo presenta el mismo problema que las NIFs: un fallo que produzca una caída del *port driver* supone también la caída del programa Erlang [1].

2.5. C Nodes

Los *C nodes* son otra opción de interoperabilidad entre C y Erlang. Este tipo de nodos son programas escritos en C que desempeñan la función de nodos ocultos en un sistema distribuido haciendo uso de la librería *Erl_Interface* para el tratamiento de los datos. Son tratados como cualquier otro nodo de Erlang, de modo que llamar a una función de C sólo se precisa enviar un mensaje al mismo, indicando mediante una tupla el nombre del proceso registrado, el nombre del C Node, y esperar a que el nodo C devuelva otro mensaje con el resultado [1].

El programa en C que desempeña la función de *C node*, haciendo uso de la librería *Erl_Interface*, inicializa el nodo y establece la comunicación entre Erlang y C. En esta conexión el *C node* puede actuar como cliente o como servidor en función de las necesidades del sistema. La recepción y el envío de mensajes dentro de este nodo se realiza también mediante el uso de la librería *Erl_Interface* [1].

Este mecanismo es de más alto nivel que los anteriores. Sin embargo, incorpora más complejidad ya que, además de lanzar el nodo C como un proceso independiente del sistema operativo, requiere la ejecución de un proceso adicional del sistema operativo (`epmd`) en segundo plano.

2.6. Comparativa de tiempos de ejecución

Una vez vistos diferentes mecanismos de interoperabilidad entre Erlang y C resulta interesante la realización de una comparativa de tiempos de ejecución que permita evaluar cuál de ellos es el más recomendable para su uso en este proyecto.

Los datos aquí mostrados han sido tomados de un artículo publicado en el blog *potatosalad* [5] en el que se realiza una comparativa de tiempos de latencia entre los distintos mecanismos tratados en este capítulo (*C nodes*, *ports*, *port drivers* y *NIFs*).

Para la realización de estas pruebas se ha implementado un test con cada uno de los mecanismos a analizar. Este test consiste en enviar y recibir un termino Erlang desde el código Erlang al código C (y viceversa) con un tamaño suficiente como para poder revelar diferencias de tiempo significativas. El término elegido para la realización de las pruebas consiste en una tupla cuyo único elemento es un binario de 64KB compuesto por 1s. El envío y recepción se realiza 100.000 veces.

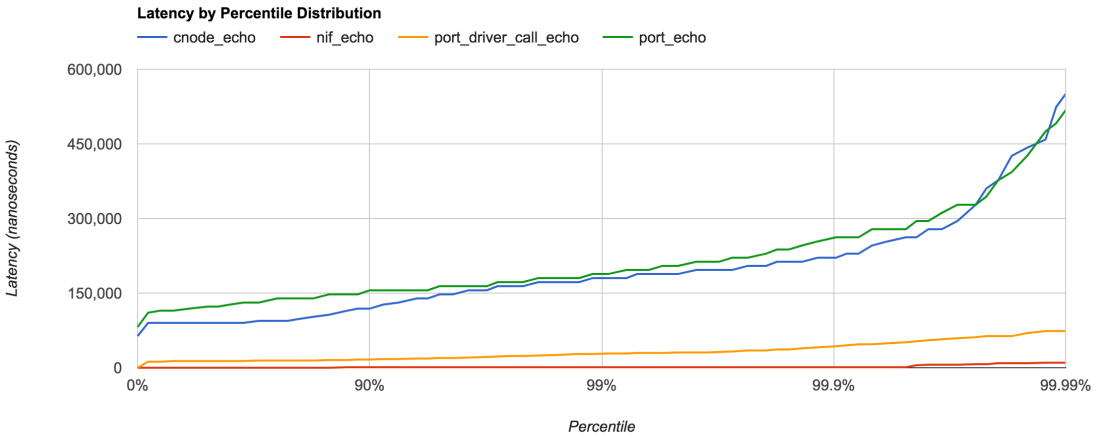

Figura 2.3: *Comparativa de latencias entre C nodes, ports, port drivers y NIFs [5]*

Como puede observarse en los resultados presentados en las figuras 2.3 y 2.4, la opción que ofrece una menor latencia son las *NIFs*, seguido muy de cerca por los *port drivers*. No obstante, como ya se ha mencionado en secciones anteriores, estos mecanismos son menos seguros, debido a que un fallo en el código C ocasionaría la caída de toda la máquina virtual

Type	Isolation	Latency
Node	Network	~100µs
Port	Process	~100µs
Port Driver	Shared	~10µs
NIF	Shared	~0.1µs

Figura 2.4: *Comparativa de latencia media entre C nodes, ports, port drivers y NIFs [5]*

de Erlang, por lo que la decisión de usar un mecanismo u otro dependerá principalmente de las necesidades propias del sistema (baja latencia o robustez).

2.7. Prueba de concepto: SDL mediante C

Antes de comenzar con la implementación del generador automático de *bindings* en Erlang se han realizado varias pruebas de concepto a modo de primer acercamiento a las tecnologías que van a ser utilizadas a lo largo del proyecto.

Figura 2.5: *Prueba de concepto implementada en C usando la librería SDL*[1]

La primera de ellas consiste en una implementación sencilla puramente en C haciendo uso de la librería para gráficos 2D SDL (*Simple DirectMedia Layer*), para la cual van a generarse los *bindings* con la herramienta final desarrollada. En esta implementación se ha hecho uso de algunas de las funciones más comunes de SDL, como pueden ser: dibujado de superficies, carga de imágenes y tratamiento de eventos.

Este test consiste en una nave espacial sobre un fondo estático (ver figura 2.5) por el que puede desplazarse mediante las flechas del teclado y realizar disparos pulsando la tecla espacio. El programa termina al pulsar la tecla ESC o bien cerrando la ventana.

El siguiente listado muestra a modo de ejemplo la función `main()` simplificada, donde puede apreciarse la inicialización y finalización de SDL así como el bucle principal donde ser realiza la captura de eventos y el dibujado de superficies.

```
int main () {
   // Inicialización de SDL y creación de la ventana
   Init()
   // Carga de imágenes
   LoadMedia()

   SDL_bool quit = SDL_FALSE;
   SDL_Event e;
   shipCurrent = shipSurface[KEY_PRESS_SURFACE_DEFAULT];

   while (!quit) {
      // Bucle de recepción de eventos
      while (SDL_PollEvent(&e) != 0) {
         // Si el evento recibido es de cierre de ventana, finalizamos
         if (e->type == SDL_QUIT) {
            quit = SDL_TRUE;
         }
         if (e->type == SDL_KEYDOWN && e->key.keysym.sym == SDLK_SPACE)
            addShot(&s1);
         HandleShipMovement(e);
      }
      MoveShip(&s1);
```

[1]Las imágenes de la nave y de los disparos empleadas pertenecen a la web OpenGameArt y han sido creadas por el usuario Skorpio (https://opengameart.org/content/space-ship-mech-construction-kit-2 y https://opengameart.org/content/sci-fi-effects, respectivamente). La imagen del fondo también ha sido tomada de OpenGameArt (https://opengameart.org/content/space-background-1).

```
        // Volcar la superficie del fondo en la pantalla
        SDL_BlitSurface(background, NULL, screen, NULL);
        // Volcar la superficie con la nave en la pantalla
        SDL_Rect shipRect = {s1.x, s1.y, shipCurrent->w*s1.sw, shipCurrent->h*
    s1.sh};
        SDL_BlitScaled(shipCurrent, NULL, screen, &shipRect);
        // Volcar los disparos en la pantalla
        drawShots(&s1, laserSurface);
        // Actualizar la superficie de la ventana
        SDL_UpdateWindowSurface(window);
    }

    // Liberar recursos
    End();

    return 0;
}
```

2.8. Prueba de concepto: SDL mediante NIFs

Una vez implementada la prueba de concepto con la librería SDL directamente en C, el siguiente paso consiste en implementar una prueba con las mismas características y funcionalidades en Erlang haciendo uso de *NIFs*.

Para el desarrollo de esta nueva prueba se han creado *NIFs* para cada una de las funciones de SDL empleadas en la implementación de C mencionada anteriormente y se ha replicado la implementación del test en Erlang haciendo uso de dichas *NIFs*.

A priori la ejecución del programa no ha arrojado problemas destacables en inicialización de SDL, el manejo de estructuras o el dibujado de imágenes y gráficos en la ventana. No obstante ha surgido un problema crítico en el manejo de eventos que ha supuesto un bloqueo en la implementación de esta prueba.

Dicho problema impide la obtención de eventos (de teclado, en este caso), ya que la función SDL que se encarga de esto (`SDL_PollEvent`) requiere ser ejecutada desde el hilo principal del programa, cosa que no ocurre siempre cuando se llama a esta función desde

una *NIF*. Al llamar a esta función desde una *NIF*, Erlang hace su propia gestión de hilos en función del sistema operativo u otros factores, de modo que no es posible asegurar que ésta sea ejecutada desde el hilo principal del programa.

Las pruebas realizadas se han ejecutado bajo un entorno macOS 10.13.6, así como un entorno Linux (Fedora 28, kernel 4.17.11). Este último no ha presentado problemas en la obtención de eventos. Sería posible solventar el problema de concurrencia obtenido mediante el manejo explícito de hilos en las *NIFs*, de manera que siempre se asegurase que las llamadas a las funciones SDL se realizasen desde un único hilo. Sin embargo, el mecanismo de comunicación entre hilos introduciría una capa de complejidad adicional que pondría en cuestión la ganancia en eficiencia obtenida mediante el uso de *NIFs* sobre otros mecanismos de interoperabilidad. Por otro lado, las *NIFs* han de ser ejecutadas en un lapso de tiempo determinado con el fin de no interferir con el planificador de procesos de Erlang. No existen garantías de que todas las funciones de la librería SDL terminen antes de dicho lapso. Por estos motivos, se ha decidido descartar el mecanismo de comunicación con *NIFs* a fin de evitar problemas e incompatibilidades futuras.

2.9. Prueba de concepto: SDL mediante Ports

Como última prueba de concepto se ha realizado algo similar a la prueba anterior, siendo en este caso los puertos el método de comunicación entre Erlang y C seleccionado.

Al igual que en el ejemplo comentado en la sección 2.3, se han implementado mecanismos de serialización y deserialización para cada uno de los tipos de datos empleados en el test (tanto en Erlang como en C), funciones en Erlang correspondientes a cada función SDL utilizada que realizan el envío y la recepción de datos a través del puerto, así como funciones manejadoras (*handler*) en C que se encargan de leer los datos obtenidos del puerto, deserializarlos, realizar la operación solicitada, serializar el resultado como una cadena de bytes, y devolverlo a Erlang.

Haciendo uso de estas implementaciones basadas en puertos se ha realizado una nueva

adaptación del programa Erlang que realiza la prueba de concepto con la nave espacial, obteniendo en este caso resultados satisfactorios. No se ha observado ningún problema para la ejecución del conjunto de funcionalidades seleccionadas para el test. Además, se ha realizado una medida de tiempos de ejecución respecto a la prueba implementada puramente en C que ha dado los siguientes resultados:

- SDL en C: ha obtenido una media aproximada de **250 fps** (fotogramas por segundo) con la nave espacial en movimiento y disparo constante.
- SDL en Erlang con puertos: en las mismas condiciones se ha obtenido una medida aproximada de **160 fps**.

En vista de estos resultados, y a pesar de que la implementación en Erlang mediante puertos supone una disminución de casi 100 fps con respecto a la versión en C, se ha concluido que este sistema es lo suficientemente eficiente como para su utilización definitiva en la herramienta de generación automática de *bindings* que se desarrolla en este proyecto.

Capítulo 3

Generación de código: tipos básicos, macros y funciones

Una vez vistas las distintas opciones de interoperabilidad entre Erlang y C, se va a pasar a describir las estrategias que se han seguido para la generación automática de *bindings*, es decir, las técnicas y procedimientos que sigue el código que genera la herramienta para tratar cada uno de los elementos que pueden encontrarse en una librería de C, con el fin de que esta pueda ser utilizada desde Erlang. Como se comenta en la sección 2.9, la comunicación entre ambos lenguajes se realiza por medio puertos de Erlang, aunque el uso de este mecanismo queda abstraído mediante las funciones generadas por esta herramienta.

Este capítulo se centra en introducir aspectos generales de la generación de los *bindings*, así como en la descripción de aspectos concretos del tratamiento de tipos básicos, *enums* y funciones.

3.1. Fichero de especificación

La herramienta de generación de *bindings* desarrollada trabaja en base a un fichero de especificación que contiene toda la información necesaria para generar tanto el código C como el código Erlang necesarios para el tratamiento de macros, tipos de datos y funciones. Este fichero establece la correspondencia entre los elementos de ambos lenguajes, y ha de ser escrito manualmente por el programador que desee adaptar una librería concreta de C

a Erlang.

El fichero de especificación presenta una estructura concreta que sigue una sintaxis en forma de una gran tupla de Erlang, con otras tuplas y elementos anidados, lo cual facilita a la herramienta su lectura y transformación en forma de `records` de Erlang. Concretamente, su estructura queda definida como puede observarse en el siguiente ejemplo, correspondiente a algunos elementos de la librería SDL:

```
{spec,
   {generator_info, "sdl_ports_gen.erl", "sdl_ports_gen.hrl", ...},
   {erlang_header, "% Erlang header code"},
   {c_header, "// C header code"},
   [
      {macro_spec, "SDL_INIT_VIDEO", "16#00000020" },
      {macro_spec, "SDL_QUIT", "16#100"},
      ...
   ],
   [
      {type_spec, uint32, "Uint32", {int, 32}, [unsigned]},
      {type_spec, uint8, "Uint8", {int, 8}, [unsigned]},
      {type_spec, window, "SDL_Window", {struct, opaque}, []},
      {type_spec, color, "SDL_Color", {struct, [
         {struct_member, r, "r", uint8, []},
         {struct_member, g, "g", uint8, []},
         {struct_member, b, "b", uint8, []},
         {struct_member, a, "a", uint8, []}
         ]}, []},
      {type_spec, palette, "SDL_Palette", {struct, [
         {struct_member, ncolors, "ncolors", int, []},
         {struct_member, colors, "colors", {pointer, color}, []},
         {struct_member, version, "version", uint32, [internal]},
         {struct_member, refcount, "refcount", int, [internal]}
         ]}, []},
      ...
   ],
   [
      {fun_spec, create_window, "SDL_CreateWindow", [
         {param_spec, string, []},
         {param_spec, int, []},
         {param_spec, int, []},
         {param_spec, int, []},
```

```
        {param_spec, int, []},
        {param_spec, uint32, []}
        ],
        {pointer, window}, []},
     {fun_spec, get_window_surface, "SDL_GetWindowSurface", [
        {param_spec, {pointer, window}, []}
        ],
        {pointer, surface}, []},
     ...
  ]
}.
```

El fichero consta de una tupla principal de 7 elementos:

- Átomo `spec`: para identificar la tupla principal de especificación.
- Tupla `generator_info`: contiene una serie de elementos de tipo *string* con información general para la herramienta:
  - Átomo que identifica el tipo de tupla (y que sirve como nombre de registro en Erlang).
  - Nombre del fichero de código Erlang generado (`.erl`).
  - Nombre del fichero de cabecera Erlang generado (`.hrl`).
  - Nombre del módulo Erlang que contendrá los métodos generados.
  - Nombre del puerto para realizar la comunicación.
  - Nombre del fichero de código C generado (`.c`).
  - Nombre del ejecutable del código C una vez sea compilado.
  - Nombre del fichero de cabecera (`.h`) de la librería de C para la cual se van a generar los *bindings*.
- Tupla `erlang_header`: contiene un átomo identificador de la tupla, seguido de una cadena de texto con el código Erlang "en bruto" que desee añadirse a la cabecera del

código generado. Por ejemplo, la definición de alguna función auxiliar o macro ajenas a la librería de C.

- Tupla `c_header`: similar a la tupla anterior pero en este caso el código se añade a la cabecera del fichero generado en C.

- Lista de tuplas `macro_spec`: cada tupla contiene la información necesaria para definir en Erlang las macros y constantes de la librería C a utilizar. Contienen los siguientes elementos en el siguiente orden: átomo identificador de la tupla (`macro_spec`), nombre de la macro/constante (`string`) y valor de la macro/constante (`string`).

- Lista de tuplas `type_spec`: en este caso cada tupla contiene la información que necesita el generador de código para tratar los tipos de datos definidos en la librería C (*structs*, *unions*, punteros, *arrays*...). La estructura de estas tuplas es la siguiente: átomo identificador de la tupla (`type_spec`), nombre del tipo de dato en Erlang (átomo), nombre del tipo de dato en C (`string`), descripción del tipo (varía en función del tipo de dato, y se describirá a lo largo de los siguientes capítulos) y una lista de opciones para indicar al generador de código ciertas particularidades (por ejemplo, si una estructura tiene en la librería C su propia función destructora).

- Lista de tuplas `fun_spec`: este último elemento consiste en una lista de tuplas que contienen la información de cada función de la librería C y están estructuradas de la siguiente manera: átomo identificador de la tupla (`fun_spec`), nombre de la función en Erlang (átomo), nombre de la función en C (`string`), lista con los parámetros que recibe la función (tupla con distintos elementos), tipo de dato que devuelve la función (es un átomo y debe ser un tipo básico, un tipo definido previamente en el fichero o `void`) y una lista de opciones para el generador (por ejemplo, si un puntero devuelto debe ser gestionado por el recolector de basura).

  En el caso del tipo devuelto por la función, si es un puntero se especifica mediante una tupla `{pointer, NombreTipo}`, siendo *NombreTipo* un átomo correspondiente al

tipo del puntero.

La lista de parámetros de la función también se define a su vez mediante tuplas con la siguiente estructura: átomo identificador de la tupla (`param_spec`), átomo con el tipo de dato del parámetro (o `{pointer, NombreTipo}` para el caso de punteros), lista de opciones para el generador de código (por ejemplo, para indicar si es un parámetro de salida).

La herramienta lee este fichero como una tupla Erlang mediante la función `io:read` y va haciendo uso de cada uno de sus elementos en distintos procedimientos para generar cada una de las partes de código necesarias para el correcto funcionamiento de los *bindings*.

3.2. Comunicación entre Erlang y C

En esta sección se va a describir el proceso de comunicación que se produce entre Erlang y C cuando se llama a cualquier método Erlang generado por la herramienta que requiera la ejecución de código C para obtener un resultado.

Inicialmente el código Erlang generado se encarga de inicializar el puerto para la comunicación. Para ello crea y registra un proceso, que denominaremos *port owner*, que se encarga de gestionar el envío y la recepción de datos entre Erlang y C. El nombre con el que se registra este proceso se indica en su apartado correspondiente en el fichero de especificación.

```
init_port() ->
   Pid = spawn(fun() -> start_port_owner("sdl_ports_gen") end),
   register(?PORT_NAME, Pid),
   ok.

start_port_owner(Name) ->
   Port = open_port({spawn, Name}, [{packet, 2}]),
   loop_port_owner(Port).
```

Una vez creado el proceso *port owner*, este inicializa el puerto mediante la función `open_port` de Erlang, indicándole la ruta del fichero ejecutable C con el que va a comunicarse y otras opciones definidas por defecto que, en este caso, indican que los paquetes de

datos enviados irán precedidos de 2 bytes indicando su tamaño.

Hecho esto, el *port owner* pasa a ejecutar un bucle (función `loop_port_owner`) que recibe el *PID* del puerto y se mantiene a la espera de peticiones de envío de mensajes a C. Las peticiones recibidas son del tipo `{Pid, {call, List}}`, donde `Pid` es el identificador del proceso Erlang que realiza la petición y al cual se devolverá el resultado, `call` es un átomo para "etiquetar" la petición, y `List` es el mensaje a enviar a C en formato de lista de bytes que contiene un código de operación y los parámetros de la función a ejecutar. Una vez recibida esta petición, el *port owner* envía la lista de bytes al puerto y espera a recibir el mensaje de respuesta, que será reenviado al proceso Erlang que realizó la petición, el cual será el que se encargue de tratarlo.

Tanto el bucle como la función para realizar las peticiones al *port owner* pueden verse en la siguiente sección de código.

```
% Esta función es ejecutada por el port owner.
loop_port_owner(Port) ->
   receive
      {Pid, {call, List}} ->
         % Se ha recibido una petición de llamada por parte del
         % proceso 'Pid'. Redirigimos esta petición al puerto.
         Port ! { self(), { command, List }},

         % Esperamos la respuesta a través del puerto, y la
         % redirigimos al proceso 'Pid' llamante.
         receive
            { _, { data, Msg }} ->

               ?DEBUG("Received binary: ~w~n", [Msg]),
               Pid ! {self(), {datalist, Msg}};
            Other ->
               ?DEBUG("Unknown response from port: ~w~n", [Other]),
               Pid ! {self(), Other}
         end,
         loop_port_owner(Port)
   end.

% Esta función es ejecutada por cualquier proceso que quiera
% solicitar al port owner una llamada.
```

```
call_port_owner(PortOwner, List) ->
   % Enviamos petición al port owner.
   PortOwner ! { self(), { call , List }},
   % Esperamos la respuesta del port owner.
   receive
      {PortOwner, X} -> X
   end.
```

Por la parte del programa C, se genera también con un bucle dentro de la función `main` que se encarga de recibir y tratar los mensajes enviados desde Erlang, así como devolver las respuestas de las operaciones solicitadas.

Este bucle puede verse en la siguiente sección de código:

```
typedef unsigned char byte;

int main() {
   byte input_buffer[BUF_SIZE], output_buffer[BUF_SIZE];

   // Leemos el primer mensaje recibido
   int len_in = read_command(input_buffer);
   size_t len_out;
   // Obtenemos el código de operación
   int code;
   read_int(input_buffer, &code);

   while (len_in > 0 && code != 0) {
      handler current_handler = handlers[code];

      // Llamamos a la función correspondiente
      (*current_handler)(input_buffer, len_in, output_buffer, &len_out);

      // Escribimos por la salida estándar el resultado de output_buffer
      write_command(output_buffer, len_out);

      // Leemos el siguiente mensaje
      len_in = read_command(input_buffer);
      read_int(input_buffer, &code);
   }
}
```

El bucle funciona de la siguiente manera:

1. Se carga en un *buffer* (`input_buffer`) el mensaje del puerto así como la longitud del mensaje (los 2 bytes iniciales) mediante la función `read_command`. Esta última función, en caso de finalización del puerto, devolverá el valor `0`.

2. A continuación, mediante `read_int`, se lee del mensaje de entrada `input_buffer` los primeros bytes que indican el código de operación que identifica la función C a ejecutar.

3. El código de operación es, en realidad, una posición dentro de un array global `handlers` que contiene punteros a las funciones manejadoras de cada una de las operaciones. Todas estas funciones tienen la misma signatura, que es la indicada en el tipo `handler`, definido a continuación:

```
typedef void (*handler)(
   byte *in,       // Buffer con los parámetros de entrada
   size_t len_in,  // Longitud del buffer con los parámetros de entrada
   byte *out,      // Buffer de salida, que almacenará el resultado
                   // de la función.
   size_t *len_out // Parámetro de salida, que indica el número de bytes
                   // escritos en el buffer de salida
);
```

   El bucle se encarga de llamar a la función manejadora correspondiente al código de operación `code`. La función manejadora, a su vez, se encarga de deserializar los parámetros contenidos en el buffer `in`, llamar a la función de librería correspondiente, y de serializar el resultado para almacenarlo en el bucle `out`. La generación de estas funciones se detalla en la sección 3.5.2.

4. Finalmente se escribe el contenido del *buffer* de salida en el puerto mediante la función `write_command` para que sea recibido en la parte de Erlang y queda a la espera de un nuevo mensaje.

El último elemento a destacar de la comunicación entre Erlang y C es el código de operación, mencionado anteriormente tanto en el bucle del *port owner* de Erlang, como en el paso 2 del bucle de C. Cada petición que se realiza desde Erlang contiene (entre otros

datos) dicho código, cuyo valor está prefijado e identifica inequívocamente a la función de la librería C que se quiere ejecutar. Cuando la petición llega a C, este código es leído y el programa es capaz de identificar la operación que debe ser ejecutada. El código debe incluirse siempre justo antes del resto de parámetros de la petición que se envía al puerto desde Erlang.

3.3. Tratamiento de tipos básicos

El tratamiento de tipos básicos para la generación de *bindings* es común a todas las librerías de C e independientemente de la librería usada y, por lo tanto, no requiere de ninguna descripción en el fichero de especificación.

Independientemente de lo que se considere o no tipos básicos en C o en Erlang, se ha hecho una selección inicial de tipos considerados como básicos teniendo en cuenta su utilidad futura dentro de los bindings generados por la herramienta, tanto en la parte de C como en la de Erlang. Esto tipos son los siguientes:

- `int`: números enteros con soporte para tamaños de 8, 16, 32 o 64 bits.
- `float`: números decimales de 32 bits.
- `double`: números decimales de 64 bits.
- `string`: cadenas de texto (en C se representan como arrays de `unsigned char`).
- `pointer`: punteros representados por enteros de 64 bits.

Para cada uno de estos tipos de datos, la herramienta genera automáticamente una serie de funciones en Erlang y en C para realizar la conversión a secuencia de bytes (y viceversa), de modo que puedan ser enviados a través del puerto.

3.3.1. Tipo `int`

En Erlang la conversión a lista de bytes se realiza creando primero una secuencia de bytes a partir del valor del entero mediante la propia sintaxis para trabajar con binarios (≪`Entero : NumeroBits`≫) que ofrece el lenguaje, y posteriormente transformando dicha secuencia binaria en una lista.

```
int_to_bytelist(Value, NBits) ->
   binary:bin_to_list(<< Value:NBits >>).

bytelist_to_int(Bytelist, NBits) ->
   << Value : NBits >> = binary:list_to_bin(Bytelist), Value.
```

En el caso de C, la conversión se realiza utilizando los operadores de desplazamiento ≪ y ≫. Como puede verse en el siguiente ejemplo, la función `read_int32` lee una lista de bytes y la convierte en un `int` y `write_int32` realiza la operación contraria:

```
// A partir de un buffer de entrada (in), lee cuatro bytes y
// construye un entero. El resultado se guarda en el parámetro
// de salida result, y se devuelve la posición del buffer tras
// el último byte leído.
byte * read_int32(byte *in, int *result) {
   byte *current_in = in;

   *result = (((int)*current_in++) << 24);
   *result = *result | (((int)*current_in++) << 16);
   *result = *result | (((int)*current_in++) << 8);
   *result = *result | ((int)*current_in++);

   return current_in;
}

// Deserializa un entero y lo convierte a una secuencia de bytes
// en formato big-endian (es decir, primero el byte más significativo)
// El resultado se guarda en el buffer out, y se incrementa el parámetro
// len tantas veces como bytes se han leído. La función devuelve la
// posición de memoria siguiente al último byte escrito en el buffer
// de salida.
byte * write_int32(int *number, byte *out, size_t *len) {
   byte *current_out = out;
```

```
    *current_out++ = *number >> 24; (*len)++;
    *current_out++ = (*number >> 16) & 255; (*len)++;
    *current_out++ = (*number >> 8) & 255; (*len)++;
    *current_out++ = *number & 255; (*len)++;

    return current_out;
}
```

Como puede verse en los listados anteriores, se ha tenido en cuenta la posibilidad de trabajar con enteros de distinto tamaño. La funciones de serialización y deserialización de enteros de Erlang reciben como parámetro un valor (`NBits`) que indica el tamaño del entero (8, 16, 32 o 64). Para el código C, en cambio, se generan distintas funciones para tratar los datos de tipo int (`read_int8`, `write_int8`, `read_int16`...) en función de su tamaño.

En el fichero de especificación se puede indicar que un tipo de dato entero tiene un tamaño determinado mediante una tupla del tipo `{int, NBits}`, siendo *NBits* el tamaño del entero en bits (8, 16, 32 o 64). Si se indica el tipo de dato `int` sin más, se toma por defecto el tamaño de 32 bits.

3.3.2. Tipo `float`

Para este tipo de dato se sigue una estrategia similar a la empleada con el tipo int en Erlang, con la particularidad de que en este caso hay que especificar el tipo `float`: ≪ `Entero : NumeroBits/float` ≫. Por defecto siempre se usa un tamaño de 32 bits con codificación IEEE 754, que es igual a la utilizada en C.

```
float_to_bytelist(Value, NBits) ->
    binary:bin_to_list(<< Value:NBits/float >>).

bytelist_to_float(Bytelist, NBits) ->
    << Value : NBits/float >> = binary:list_to_bin(Bytelist), Value.
```

En C la conversión se ha realizado utilizando estructuras `union`, que contienen una variable de tipo `float` así como una de tipo array de `unsigned char`, permitiendo que al utilizar una u otra la conversión se realiza de forma automática, ya que estas ocupan el mismo espacio de memoria.

```
byte * read_float(byte *in, float *result) {
   byte *current_in = in;
   union {
      float number;
      unsigned char arr[4];
   } aux_float;

   // Escribimos en el union los bytes de entrada
   for (int i=3; i>=0; i--)
      aux_float.arr[i] = *current_in++;

   // Leemos el float resultante.
   *result = aux_float.number;

   return current_in;
}

byte * write_float(float *number, byte *out, size_t *len) {
   byte *current_out = out;
   union {
      float number;
      unsigned char arr[4];
   } aux_float;

   // Escribimos en el union el float de entrada
   aux_float.number = *number;

   // Obtenemos los bytes de este mismo union.
   for (int i = 3; i >= 0; i--)
      *current_out++ = aux_float.arr[i];

   return current_out;
}
```

3.3.3. Tipo **double**

El tipo double en Erlang se trata como un float de 64 bits, de modo que sólo se necesita llamar a las funciones de tratamiento de floats (sec. 3.3.2) especificando el tamaño de 64 bits. De nuevo, la codificación utilizada por Erlang y C es el estándar IEEE 754.

```
double_to_bytelist(Value) ->
   float_to_bytelist(Value, 64).

bytelist_to_double(Bytelist) ->
   bytelist_to_float(Bytelist, 64).
```

Para su tratamiento en C se ha seguido la misma estrategia que con el tipo `float` (sec. 3.3.2) utilizando una estructura `union`. Puede verse en el siguiente ejemplo de la función `read_double`, que realiza la conversión de lista de bytes a `double`:

```
byte * read_double(byte *in, double *result) {
   byte *current_in = in;
   union {
      double number;
      unsigned char arr[8];
   } aux_double;

   // Escribimos en el union los bytes de entrada
   for (int i = 7; i >= 0; i--)
      aux_double.arr[i] = *current_in++;

   // Y leemos del mismo union el byte de salida
   *result = aux_double.number;

   return current_in;
}
```

3.3.4. Tipo `string`

En el caso de los `string`, la estrategia que se ha seguido para la conversión en lista de bytes en Erlang consiste en la utilización de la función `unicode:characters_to_binary`, que se encarga de convertir una cadena de texto en una secuencia de bytes, que a su vez será convertida en una lista. Para que C pueda interpretar esta cadena se añade al final el carácter nulo de terminación de cadena (`\0`). Este método permite indicar la codificación del texto, siendo la opción por defecto *UTF-8*.

```
string_to_bytelist(Value, Encoding) ->
   binary:bin_to_list(unicode:characters_to_binary(Value, Encoding))++[$\0].
```

```
bytelist_to_string(Bytelist, Encoding) ->
   unicode:characters_to_list(binary:list_to_bin(Bytelist), Encoding).
```

En la parte de C, las cadenas de texto se van leyendo byte a byte del *buffer* de entrada del puerto hasta encontrar el carácter nulo. De igual modo, para pasar cadenas de texto (arrays de `unsigned char`) como secuencias de bytes por el *buffer* de salida del puerto se van escribiendo uno a uno los caracteres de la cadena y finalmente se añade el carácter nulo.

```
typedef unsigned char string[BUF_SIZE];

byte * read_string(byte *in, string *result) {
   byte *current_in = in;

   int i = 0;

   // Copiamos del buffer in al array con la cadena result hasta que
   // encontremos el carácter nulo, o excedamos el tamaño del array
   // result.
   while (*current_in != '\0' && i<(BUF_SIZE-1)) {
      (*result)[i] = *current_in;
      i++; current_in++;
   }

   // Si hemos excedido el tamaño del array result, aun así hemos
   // de llegar hasta el carácter nulo del buffer de entrada, para
   // devolver la posición inmediatamente posterior.
   if (i >= (BUF_SIZE-1)) {
      while (*current_in != '\0')
         current_in++;
   }

   // Agregamos el caracter nulo al array resultante.
   (*result)[i] = '\0'; current_in++;

   return current_in;
}

byte * write_string(string *str, byte *out, size_t *len) {
   byte *current_out = out;

   // Obtenemos la longitud de la cadena a escribir
```

```
   int strsize = (int) strlen((const char *) *str);

   // Copiamos el resultado en el buffer de salida
   for (long i = 0; i < strsize; i++) {
      *current_out++ = (*str)[i]; (*len)++;
   }
   *current_out++ = '\0'; (*len)++;

   return current_out;
}
```

3.3.5. Tipo `pointer`

El tratamiento de punteros en Erlang se ha realizado como si se tratara de enteros de 64 bits con formato `big-endian`. Por lo tanto, la conversión a lista se bytes se basa en llamar a las funciones definidas para convertir enteros (sec. 3.3.1), indicando un tamaño de 64 bits.

```
pointer_to_bytelist(Value) ->
   int_to_bytelist(Value, 64).

bytelist_to_pointer(Bytelist) ->
   bytelist_to_int(Bytelist, 64).
```

Algo similar ocurre en C. Los punteros son tratados por las funciones definidas para enteros de 64 bits. El valor entero en secuencia de bytes se lee de mediante la función `read_int64`, almacenando su valor en una variable de tipo `uintptr_t`. Los tipos enteros `intptr_t` y `uintptr_t` están pensados para almacenar punteros, por lo tanto la conversión entre punteros y este tipo especial de enteros siempre es posible [8]. En el proceso contrario, el puntero se convierte a `uintptr_t`, y este se escribe en el *buffer* de salida del puerto como entero de 64 bits.

```
byte * read_pointer(byte *in, void **result) {
   byte *current_in = in;
   uintptr_t p;

   current_in = read_int64(current_in, (int64_t *) &p);
   *result = (void *) p;
```

```
    return current_in;
}

byte * write_pointer(void **pointer, byte *out, size_t *len) {
    byte *current_out = out;
    uintptr_t p = (uintptr_t) (*pointer);

    current_out = write_int64((int64_t *) &p, current_out, len);

    return current_out;
}
```

3.3.6. Funciones auxiliares de deserialización

Adicionalmente se han añadido en el fichero generado, para cada tipo, unas funciones denominadas `parse_XXX`, donde `XXX` es un tipo de dato básico (`int`, `float`, `string`...). Estas funciones se encargan convertir una lista de bytes al tipo en cuestión, con la suposición de que la lista de entrada no tiene por qué tener una longitud exactamente igual a la requerida por el dato a deserializar, sino que puede contener bytes adicionales tras el mismo. De este modo, las funciones `parse_XXX`, tras realizar la conversión, devuelven una tupla que contiene el dato en cuestión convertido y el resto de la lista de bytes "sobrante". La siguiente sección de código muestra, a modo de ejemplo, el código de la función `parse_int`.

```
parse_int(Bytelist, NBytes) ->
    parse_int(Bytelist, NBytes, NBytes, []).
parse_int([B|Rest], NBytes, Cnt, Result) when Cnt>0 ->
    parse_int(Rest, NBytes, Cnt-8, [B|Result]);
parse_int(Bytelist, NBytes, _Cnt, Result) ->
    {bytelist_to_int(lists:reverse(Result), NBytes), Bytelist}.
```

Este tipo de funciones resultan muy útiles cuando se tiene una lista de bytes que contiene varios datos concatenados y se quiere ir haciendo la conversión de cada uno de ellos llamando secuencialmente a sus correspondientes funciones `parse_XXX`.

3.4. Tratamiento de macros

Vistos los tipos básicos, el siguiente elemento a tratar son las constantes y macros. Las librerías de C, como en este caso SDL, contienen numerosas macros y constantes que en ocasiones deben ser utilizadas por el programador para pasarlas como parámetro en una función, o para otro tipo de usos. De este modo, si se pretende usar una librería de C desde Erlang, se debe disponer de este tipo de elementos.

En este caso se ha optado por que el programador indique, en el fichero de especificación, las macros y constantes en la librería de C, junto con su valor correspondiente (sec. 3.1), de modo que el generador de código defina una macro en Erlang (mediante el uso de la directiva `-define`) por cada macro definida en la librería C.

Por ejemplo, supongamos que se tiene en el fichero de especificación la definición de las siguientes macros de la librería SDL:

```
{spec,
   ...
   [
      {macro_spec, "SDL_INIT_VIDEO", "16#00000020" },
      {macro_spec, "SDL_QUIT", "16#100"},
      ...
   ],
    ...
}.
```

La herramienta recorre la lista de macros y, para cada una de ellas, genera una macro en Erlang con el nombre y valor definidos en el fichero, dando como resultado el siguiente bloque de código:

```
-define(SDL_INIT_VIDEO, 16#00000020).
-define(SDL_QUIT, 16#100).
```

Cabe puntualizar que el generador de código no realiza la definición de estas macros en el fichero principal de código Erlang generado (`.erl`), sino en un fichero de cabecera (`.hrl`), al cual se hace referencia dentro del fichero principal.

3.5. Tratamiento de funciones

La generación de funciones es uno de los apartados principales para la generación de los *bindings*, ya que, de todos los elementos de código que se generan, estos son con los que el programador va a interactuar de forma más directa en el código Erlang.

Para la ejecución de funciones de C desde Erlang se necesita generar: una función en Erlang que realice la petición correspondiente a C a través del puerto y espere el resultado de la operación solicitada, y una función en C que se encargue de recoger esa petición para ejecutar dicha operación y devolver el resultado a Erlang también a través del puerto.

Supongamos que se tiene en el fichero de especificación la siguiente función de SDL `SDL_LoadBMP`, que se encarga de cargar una imagen en formato `.bmp` recibiendo como único parámetro el *path* del fichero (`string`). Como se comenta en la sección 3.1, la información presente en cada especificación de función es:

```
{fun_spec,
   nombre_erlang,
   nombre_c,
   lista_parámetros,
   tipo_devuelto,
   lista_opciones}
```

Y a su vez, dentro de `lista_parametros`, se encuentran tuplas que definen cada parámetro de la función con la siguiente información:

```
{param_spec,
   tipo_dato,
   opciones}
```

Teniendo en cuenta esto, el ejemplo antes comentado presenta en el fichero de especificación una apariencia similar a la que se aprecia en el siguiente fragmento:

```
{spec,
   ...
   [
      {fun_spec, load_bmp, "SDL_LoadBMP", [
         {param_spec, string, [const]}],
         {pointer, surface}, []},
```

```
      ...
   ]
}.
```

Una vez que la herramienta ha leído la lista de funciones se encarga de generar el código para cada una de ellas. Primero genera el código en Erlang y a continuación hace lo propio para C. Este orden se mantiene para el resto de elementos generados (macros, tipos, etc.).

3.5.1. Generación de código Erlang

Centrándonos en el código generado en Erlang, cada función se genera mediante el siguiente proceso:

1. La herramienta lee la especificación de los parámetros que recibe la función (obtenidos a partir de `lista_parametros`) y genera una cabecera con el nombre especificado para la funcion Erlang (`nombre_erlang`), seguido de los parametros a los que le asigna un nombre por defecto en función de su tipo (p.e. *String_1*, *Int_2*...). En este caso la cabecera resultante es: `load_bmp(String_1)`.

2. A continuación se generan una serie de líneas de código para convertir tanto el código de operación como cada uno de los parámetros de la función en listas de bytes. Para ello se hace uso de las funciones correspondientes que realizan esta serialización para cada tipo de dato básico, que habrán sido generadas con anterioridad. Como la función `load_bmp` solo recibe un parámetro *string*, en este caso solo se generan dos líneas, una para serializar el código de operación (en el caso de `load_bmp` este código es el 761) y otra para serializar dicho *string*.

3. El siguiente paso es generar la petición al *port owner* con las listas de bytes generadas previamente así como la espera de los resultados mediante una expresión `case`. Aquí se contempla la posibilidad de recibir el resultado esperado (caso `{datalist, DataList}`) o un error (caso `Msg`).

4. En caso de recibir la respuesta esperada, se genera una linea por cada resultado obtenido. Esta línea llama a una de las funciones `parse_XXX` (también generada con anterioridad, ver sección 3.3.6) que se encargará de convertir la lista de bytes al tipo básico correspondiente. Como en este caso la función solo devuelve un puntero sólo se llama a la función `parse_pointer` y se devuelve el resultado. En el caso de tener varios elementos de retorno (por ejemplo, en funciones con parámetros de salida) se encapsulan dentro de una tupla que se devuelve como resultado final.

5. En el caso de obtener un mensaje de error solo se devuelve la tupla `{error, Msg}`.

```
load_bmp(String_1) ->
   Code = int_to_bytelist(761),
   Param1 = string_to_bytelist(String_1),
   ResultCall = call_port_owner(?PORT_NAME, [Code, Param1]),
   case ResultCall of
      {datalist, DataList} ->
         {RetParam1, _R1} = parse_pointer(DataList),
         RetParam1;
      Msg ->
         {error, Msg}
   end.
```

En el caso de la función tenga parámetros de salida, como por ejemplo `SDL_GetWindowSize`, se incluye en la lista de opciones del parámetro en cuestión el átomo `return`. Esto es interpretado por la herramienta generadora, que omite estos parámetros en la fase de conversión y envío al puerto (fases 2 y 3), y se encarga de recibirlos como resultado de la operación (fase 4).

A continuación puede verse la parte del fichero de especificación correspondiente a la función `SDL_GetWindowSize`, así como el código generado por la herramienta a partir de él.

```
{fun_spec, get_window_size, "SDL_GetWindowSize", [
   {param_spec, {pointer, window}, []},
   {param_spec, {pointer, int}, [return]},
   {param_spec, {pointer, int}, [return]}
   ],
   void, []},
```

```
get_window_size(P_Window_1) ->
   Code = int_to_bytelist(767),
   Param1 = pointer_to_bytelist(P_Window_1),
   ResultCall = call_port_owner(?PORT_NAME, [Code, Param1]),
   case ResultCall of
      {datalist, DataList} ->
         R0 = DataList,
         {RetParam1, R1} = parse_int(R0),
         {RetParam2, _R2} = parse_int(R1),
         {RetParam1, RetParam2};
      Msg ->
         {error, Msg}
   end.
```

3.5.2. Generación de código C

Para la generación de las funciones en C la herramienta realiza los siguientes pasos:

1. En primer lugar se genera la cabecera de la función manejadora (siempre de tipo `void`) con el nombre especificado (`nombre_erlang`) seguido de `_Handler`. Los parámetros de estas funciones manejadoras de C siempre son los mismos: los *buffers* de entrada y salida, y sus respectivos tamaños. En el caso de ejemplo la cabecera de la función queda de la siguiente manera: `void load_bmp_Handler(byte *in, size_t len_in, byte *out, size_t *len_out)`. Además se añaden las primeras líneas de código, que también son comunes para todas las funciones manejadoras, que se encargan de inicializar los *buffers* de entrada y salida, y "saltar" en el *buffer* de entrada los primeros 4 bytes con el código de operación, que aquí ya no es de utilidad.

2. A continuación se genera el código para leer cada uno de los parámetros de la operación solicitada del *buffer* de entrada y almacenarlos en variables de sus correspondientes tipos. Al igual que en la función de Erlang, si alguno de estos parámetros es de salida (tiene incluida la opción `return`) no se generará una linea para leer su valor del *buffer*, pero si se declarará una variable para pasarla como parámetro en la función solicitada

y devolverla posteriormente a Erlang con el resultado. En este caso, como solo se tiene un parámetro de tipo `string`, se crea una variable de dicho tipo (`var1`) y se lee su valor mediante la función `read_string`.

3. Se genera el código que ejecuta la función solicitada por Erlang, que pertenece a la librería para la cual se están generando los *bindings* (`SDL_LoadBMP` en este caso), pasándole los parámetros leídos anteriormente del *buffer* de entrada.

4. Finalmente se genera la escritura en el *buffer* de salida del resultado de la función (en caso de haberlo) y de los parámetros de salida (en caso de haberlos). Para el ejemplo que estamos siguiendo se genera una línea que escribe en el puntero devuelto por la función `SDL_LoadBMP` mediante `write_pointer`.

```
void load_bmp_Handler(byte *in, size_t len_in, byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in+=4;

   string var1;
   current_in = read_string(current_in, &var1);

   SDL_Surface *retvar = SDL_LoadBMP(var1);
   current_out = write_pointer((void **) &retvar, current_out, len_out);
}
```

Con este código generado (tanto en la parte de Erlang como en la de C) ya podríamos hacer uso de la función `SDL_LoadBMP` de SDL desde la shell de Erlang, como puede verse en el siguiente ejemplo donde se carga el módulo de Erlang generado, se inicializa el puerto de comunicación con el programa C y se llama a la función `load_bmp` generada. El número final obtenido corresponde al puntero a la imagen que ha sido cargada por la función `SDL_LoadBMP` y que se encuentra almacenada en el proceso C.

```
Eshell V9.3  (abort with ^G)

1> c(sdl_ports_gen).
{ok,sdl_ports_gen}
2> sdl_ports_gen:init_port().
```

```
sdl_port initialized.
ok
3> Img = sdl_ports_gen:load_bmp("resources/img/space_bg.bmp").
140496454730592
```

Resumiendo, en este capítulo se ha comenzado a describir los aspectos principales de la generación de *bindings* para el uso de librerías de C en programas Erlang, tomando como punto de partida la librería SDL. Se ha descrito el fichero de especificación donde se describen todos los elementos de la librería a tratar por la herramienta generadora de código, así como el tratamiento de los tipos básicos y las funciones. Con esto se ha podido generar automáticamente un conjunto de *bindings* para muchas de funciones de la librería SDL que solo hacen uso de estos elementos, pero para el resto es necesario que la herramienta sea capaz de trabajar con tipos de datos más complejos (como por ejemplo estructuras, uniones o *arrays*). Estos elementos serán tratados en los siguientes capítulos.

Capítulo 4

Generación de código: *enums*, *structs* y *unions*

En este capítulo vamos a seguir describiendo aspectos de la generación de código para otro tipo de elementos, como son los *enums*, los *structs* y los *unions*.

Estos elementos son definidos, dentro del fichero de especificación (ver sección 3.1), en la lista que representa los tipos de datos contenidos en la librería C que se está tratando. A modo de recordatorio, la información de un tipo de dato se representa de la siguiente manera:

```
{type_spec,
   nombre_erlang,
   nombre_c,
   descripcion_tipo,
   lista_opciones}
```

El campo `descripcion_tipo` es donde se definen las particularidades de los *enums*, *structs* y *unions*. Cada uno de ellos presenta una estructura diferente que será descrita en las siguientes secciones dedicadas.

4.1. Tratamiento de *enums*

Los *enums* son elementos del lenguaje C que permiten definir un conjunto ordenado de constantes de tipo entero en el que se definen los distintos valores que puede tomar tener dicho elemento [3].

Como ya se ha comentado al inicio de este capítulo, la definición de un tipo de dato *enum* dentro del fichero de especificación presenta la misma estructura que cualquier otro tipo con la particularidad de que el campo `descripcion_tipo` tiene la siguiente estructura para representar la información:

```
{enum, [
   {nombre_constante_erl1, nombre_constante_c1, valor1},
   {nombre_constante_erl2, nombre_constante_c2, valor2},
   ...
   ]}
```

El átomo `enum` viene seguido de una lista de tuplas `{nombre_erl, nombre_c, valor}` en las que se hace corresponder el nombre de la constante en Erlang con su contraparte en C. Cada una de estas constantes en C lleva asociada un valor numérico. Dicho valor se encuentra en la tercera componente de la tupla.

Para ejemplificar un tipo de dato *enum* vamos a tomar `SDL_Scancode` (solo algunos de sus elementos, ya que tiene más de 200), que es un tipo *enum* definido en la librería SDL. Este tipo *enum* es utilizado para asignar un código a cada tecla del teclado. Su definición en la librería SDL es la siguiente:

```
typedef enum {
   SDL_SCANCODE_UNKNOWN = 0,
   SDL_SCANCODE_A = 4,
   SDL_SCANCODE_B = 5,
   ...
} SDL_Scancode;
```

La definición de este tipo de datos en el fichero de especificación tiene el siguiente aspecto:

```
{spec,
   ...
   [
      {type_spec, scancode, "SDL_Scancode",
         {enum, [
            {scancode_unknown, "SDL_SCANCODE_UNKNOWN", 0},
            {scancode_a, "SDL_SCANCODE_A", 4},
            {scancode_b, "SDL_SCANCODE_B", 5},
            ...
```

```
            ]
        }, []},
    ...
  ]
  ...
}.
```

4.1.1. Generación de código Erlang

El objetivo es representar los *enum* como átomos en Erlang, de modo que para ser enviados a C se necesita una función que "traduzca" estos átomos a su valor entero y poder ser enviado por el puerto. De igual modo, cuando se recibe un *enum* desde C, se recibe como un dato de tipo entero que también deberá ser traducido a átomo para ser usado en Erlang.

La herramienta genera por cada tipo *enum* un par de funciones para realizar la traducción de átomo a entero y viceversa. En este caso serían: `scancode_get_int` y `scancode_get_atom`.

```
scancode_get_int(Atom) ->
  case Atom of
    scancode_unknown -> 0;
    scancode_a -> 4;
    scancode_b -> 5;
    % ...
    _ -> undefined
  end.

scancode_get_atom(Int) ->
  case Int of
    0 -> scancode_unknown;
    4 -> scancode_a;
    5 -> scancode_b;
    % ...
    _ -> undefined
  end.
```

Una vez definidas estas funciones la herramienta genera las funciones de serialización y deserialización (ya introducidas en el capítulo anterior para los tipos básicos) con el fin de convertir un valor de tipo `scancode` a lista de bytes y viceversa. Estas funciones deben estar presentes en todos los tipos de datos, que harán uso de estas y de las funciones para

convertir enteros, ya que al final lo que se envía y se recibe desde C es un número entero representando el valor del tipo enumerado.

```
scancode_to_bytelist(Value) ->
   Int = scancode_get_int(Value),
   int_to_bytelist(Int).

bytelist_to_scancode(Bytelist) ->
   Int = bytelist_to_int(Bytelist),
   scancode_get_atom(Int).

parse_scancode(Bytelist) ->
   {Int, RList} = parse_int(Bytelist),
   {scancode_get_atom(Int), RList}.
```

4.1.2. Generación de código C

El código generado en C para *enums* es bien sencillo. Solo se necesita generar funciones para leer y escribir el el puerto los tipos *enums*, que al ser tratados como tipos `int` se limitan a llamar a las funciones que tratan este tipo de dato.

```
byte * read_scancode(byte *in, SDL_Scancode *result) {
   return read_int(in, (int *) result);
}

byte * write_scancode(SDL_Scancode *value, byte *out, size_t *len) {
   return write_int((int *) value, out, len);
}
```

Una vez generado el código para tratar tipos enumerados, podemos ver un ejemplo de su uso en la *shell* de Erlang donde se realiza la serialización de un dato de tipo `SDL_Scancode`.

```
Eshell V9.3  (abort with ^G)

1> SC_Bytelist = sdl_ports_gen:scancode_to_bytelist(scancode_a).
[0,0,0,4]
2> sdl_ports_gen:bytelist_to_scancode(SC_Bytelist).
scancode_a
```

4.2. Tratamiento de *structs*

Las estructuras (`struct`) en C permiten definir tipos de datos formados por un conjunto de elementos que pueden ser a su vez de cualquier tipo y guardan entre sí alguna relación a nivel conceptual [3].

Al igual que con los tipos *enum*, los *struct* tienen su propio tipo de definición en el campo `descripcion_tipo` del fichero de especificación:

```
{struct, [
   {struct_member, nombre_erlang1, nombre_c1, tipo1, lista_opciones1},
   {struct_member, nombre_erlang2, nombre_c2, tipo2, lista_opciones2},
   ...
   ]}
```

En este caso vamos a tomar como ejemplo la estructura `SDL_Rect`, que representa un rectángulo definido por cuatro variables `int` (coordenada x, coordenada y, ancho y alto). Su definición en SDL es la siguiente:

```
typedef struct SDL_Rect {
   int x, y;
   int w, h;
} SDL_Rect;
```

Su especificación en el fichero quedaría de la siguiente manera:

```
{spec,
   ...
   [
      {type_spec, rect, "SDL_Rect",
         {struct, [
            {struct_member, x, "x", int, []},
            {struct_member, y, "y", int, []},
            {struct_member, w, "w", int, []},
            {struct_member, h, "h", int, []}
            ]
         }, []},
      ...
   ]
   ...
}.
```

4.2.1. Generación de código Erlang

La generación de código para tratar estructuras en Erlang comienza por representarlas haciendo uso de *records*, por su cierta similitud con los *struct* de C, ya que los *records* permiten definir registros con una serie de elementos a los que se puede acceder mediante su nombre [1]. La herramienta genera un *record* para cada *struct* con los mismos campos que en C y este podrá ser utilizado en el resto de funciones generadas, así como por parte del programador. El *record* es generado en el fichero de cabecera (`.hrl`).

```
-record(rect, {x, y, w, h}).
```

Definido el *record* la herramienta genera automáticamente, como en cualquier otro tipo, las funciones de conversión a lista de bytes, que en este caso hacen uso del registro previamente definido y de las funciones de conversión de cada uno de los tipos de datos de sus elementos, en este caso 4 números enteros.

```
rect_to_bytelist(Value) ->
   Return = [int_to_bytelist(Value#rect.x),
   int_to_bytelist(Value#rect.y),
   int_to_bytelist(Value#rect.w),
   int_to_bytelist(Value#rect.h)],
   lists:flatten(Return).

bytelist_to_rect(Bytelist) ->
   R0 = Bytelist,
   {X, R1} = parse_int(R0),
   {Y, R2} = parse_int(R1),
   {W, R3} = parse_int(R2),
   {H, _} = parse_int(R3),
   #rect{x=X, y=Y, w=W, h=H}.

parse_rect(Bytelist) ->
   R0 = Bytelist,
   {X, R1} = parse_int(R0),
   {Y, R2} = parse_int(R1),
   {W, R3} = parse_int(R2),
   {H, R4} = parse_int(R3),
   {#rect{x=X, y=Y, w=W, h=H}, R4}.
```

Finalmente, la herramienta genera un par de funciones *get* y *set* para cada componente del *struct*. Estas funciones sirven para obtener y modificar los elementos de un *struct* almacenado en el proceso C, en aquellos casos en los que desde Erlang solo se dispone de un puntero al mismo. En este sentido, estas funciones reproducen el comportamiento del operador flecha (`->`) de C.

Estos pares de funciones se generan por cada elemento de cada estructura y realizan su tarea de la siguiente manera:

- `XXX_get_YYY` (donde XXX es el nombre de la estructura e YYY es el nombre del elemento): Recibe como parámetro el puntero a la estructura, lo serializa junto con el código de operación, realiza la petición a través del $port\_owner$ y recibe el elemento YYY solicitado.

- `XXX_set_YYY`: Recibe como parámetros el puntero a la estructura, así como el valor a asignar al elemento YYY, los serializa junto al código de operación y realiza la petición para modificar el valor de dicho elemento.

Para el caso del `SDL_Rect`, se generan 4 funciones *get* y 4 funciones *set*, ya que es una estructura que cuenta con cuatro elementos. A modo de ejemplo puede verse el código generado para el elemento `x` de tipo entero:

```
rect_get_x(Pointer) ->
   % El código de operación asignado para get_x es el 172
   Code = int_to_bytelist(172),
   PList = pointer_to_bytelist(Pointer),
   % Enviamos al port owner el código de operación seguido del
   % puntero a la estructura y esperamos la respuesta
   ResultCall = call_port_owner(?PORT_NAME, [Code, PList]),
   case ResultCall of
      {datalist, DataList} ->
         bytelist_to_int(DataList);
      Msg ->
         {error, Msg}
   end.
```

```
rect_set_x(Pointer, Attrib) ->
   % El código de operación asignado para set_x es el 173
   Code = int_to_bytelist(173),
   PList = pointer_to_bytelist(Pointer),
   AList = int_to_bytelist(Attrib),
   % Enviamos al port owner el código de operación seguido del
   % puntero a la estructura y el nuevo valor a asignar, y
   % esperamos la respuesta (que en este caso será vacía)
   ResultCall = call_port_owner(?PORT_NAME, [Code, PList, AList]),
   case ResultCall of
      {datalist, _DataList} ->
         ok;
      Msg ->
         {error, Msg}
   end.
```

4.2.2. Generación de código C

En el código C estas estructuras ya se encuentran definidas en la librería en cuestión, de modo que se comienza generando las funciones *read* y *write* que se encargan de ir leyendo/escribiendo del *buffer* de entrada/salida los distintos elementos del `struct`, en este caso `SDL_Rect`.

```
byte * read_rect(byte *in, SDL_Rect *result) {
   byte *current_in = in;

   current_in = read_int(current_in, &(result->x));
   current_in = read_int(current_in, &(result->y));
   current_in = read_int(current_in, &(result->w));
   current_in = read_int(current_in, &(result->h));

   return current_in;
}

byte * write_rect(SDL_Rect *value, byte *out, size_t *len) {
   byte *current_out = out;

   current_out = write_int(&(value->x), current_out, len);
   current_out = write_int(&(value->y), current_out, len);
   current_out = write_int(&(value->w), current_out, len);
```

```
   current_out = write_int(&(value->h), current_out, len);

   return current_out;
}
```

Para finalizar, se generan las funciones *Handler* que atienden las peticiones de los *get* y *set* de Erlang para cada elemento de las estructura.

- `XXX_get_YYY_Handler` (donde XXX es el nombre de la estructura e YYY es el nombre del elemento): Lee del *buffer* entrada el puntero a la estructura XXX y a partir de este escribe en el *buffer* de salida el elemento YYY solicitado mediante el operador `->`.

- `XXX_set_YYY_Handler`: Lee del *buffer* entrada el puntero a la estructura XXX así como el valor a asignar al elemento YYY y realiza la asignación mediante el operador `->`.

Siguiendo el ejemplo de `SDL_Rect`, el código generado en C para el elemento `x` es el siguiente:

```
void rect_get_x_Handler(byte *in, size_t len_in,
                        byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in+=4;

   SDL_Rect *ptr;
   current_in = read_pointer(current_in, (void **) &ptr);
   current_out = write_int(&(ptr->x), current_out, len_out);
}

void rect_set_x_Handler(byte *in, size_t len_in,
                        byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in+=4;

   SDL_Rect *ptr;
   current_in = read_pointer(current_in, (void **) &ptr);
   current_in = read_int(current_in, &(ptr->x));
}
```

En el siguiente fragmente de la shell de Erlang podemos ver un ejemplo de uso de la estructura `SDL_Surface` mediante las funciones generadas. En este caso la estructura se obtiene a partir de la función`load_bmp`, cargando una imagen de 1280x800 píxeles.

```
3> Img = sdl_ports_gen:load_bmp("resources/img/space_bg.bmp").
140496454730592
4> sdl_ports_gen:surface_get_w(Img).
1280
5> sdl_ports_gen:surface_get_h(Img).
800
```

4.3. Tratamiento de *unions*

Las uniones de C (`union`), del mismo modo que las estructuras, permiten definir tipos de datos que representan un conjunto de elementos de diversos tipos. La diferencia con estas primeras reside en que los elementos de los *union* comparten el mismo espacio de memoria, de modo que solo se puede almacenar el valor de uno de ellos, siendo éste sobrescrito si se asigna valor a cualquier otro [3].

La descripción de este tipo de datos en el fichero de especificación se realiza de la misma manera que las estructuras. De este modo, si tomamos el *union* `SDL_Event` de SDL, su especificación quedaría como puede verse a continuación.

```
{spec,
   ...
   [
      {type_spec, event, "SDL_Event",
         {union, [
            {struct_member, type, "type", uint32, []},
            {struct_member, common, "common", common_event, []},
            {struct_member, window, "window", window_event, []},
            {struct_member, key, "key", keyboard_event, []},
            ...
            ]
         }, []},
      ...
   ]
   ...
```

```
}.
```

Aunque a priori parezca que este tipo de datos pueden ser tratados de forma similar a los *struct* la realidad es diferente. La imposibilidad de saber en tiempo de ejecución el tipo de elemento que el *union* tiene almacenado en un momento concreto dificulta su serialización y envío a través del puerto. Por este motivo, se ha optado por tratar estas estructuras exclusivamente mediante punteros, es decir, a través del puerto solo puede transmitirse un puntero que haga referencia a una unión que haya sido inicializada y almacenada en el programa C. En el capítulo 5 se trata con mayor profundidad el tratamiento de punteros.

4.3.1. Generación de código Erlang

La generación de código Erlang comienza con las funciones para para convertir los *union* en listas de bytes y viceversa. Como van a ser tratados como punteros, estas funciones simplemente llaman a las funciones de serialización de punteros.

```
event_to_bytelist(Value) ->
   pointer_to_bytelist(Value).

bytelist_to_event(Bytelist) ->
   bytelist_to_pointer(Bytelist).

parse_event(Bytelist) ->
   parse_pointer(Bytelist).
```

Además de éstas, también se generan funciones *get* y *set* para cada elemento del `union`, cuya estructura y funcionalidad es exactamente igual a la descrita para *structs* (sec. 4.2).

4.3.2. Generación de código C

Al igual que en Erlang, la generación en C se limita a las funciones *read* y *write*, que leen y escriben respectivamente del puerto el puntero que hace referencia al `union` para que pueda ser utilizado por otras funciones, así como las funciones *Handler* que atienden las peticiones de los *get* y *set* (también de estructura y funcionalidad similar a los *structs*).

Aquí se muestra un pequeño ejemplo de las funciones *read* y *write* generadas para el tipo `SDL_Event`:

```
byte * read_event(byte *in, SDL_Event *result) {
   return read_pointer(in, (void **) &result);
}

byte * write_event(SDL_Event *value, byte *out, size_t *len) {
   return write_pointer((void **) &value, out, len);
}
```

4.4. Tratamiento de *structs* y *unions* opacos

Existe un caso especial para el cual no es válida la forma de proceder con *structs* y *unions* descrita en las secciones anteriores: las estructuras y uniones opacas.

El lenguaje C ofrece la posibilidad de declarar tipos de datos abstractos, es decir, declarar *struct* y *union* sin definir los elementos que los componen. Como se menciona en [15], haciendo uso del mecanismo de *headers* (`.h`) y código (`.c`) se separa la interfaz de estos tipos de datos de su implementación, de modo que sus elementos solo son accesibles mediante punteros y funciones de acceso y modificación.

Un ejemplo de este tipo de estructuras en la librería SDL es el tipo `SDL_Window`, que representa una ventana de SDL donde se representan los elementos gráficos. Su definición puede verse a continuación.

```
typedef struct SDL_Window SDL_Window;
```

La definición completa de este tipo de dato se encuentra en un fichero aparte que no tiene por qué conocer ningún usuario de la librería, de modo que solo es posible acceder a sus elementos mediante las funciones proporcionadas por la propia librería SDL a través de un puntero a `SDL_Window` (p.e. `SDL_GetWindowSize` para obtener el ancho y alto de la ventana).

Los elementos opacos se representan en el fichero de especificación de la misma forma que cualquier *struct* o *union*, sustituyendo la lista de elementos por el átomo `opaque`. Podemos

ver la especificación de SDL_Window en el siguiente fragmento:

```
{type_spec, window, "SDL_Window", {struct, opaque}, []}
```

Desde el punto de vista de la generación de código, estas estructuras son tratadas como punteros. Tanto las funciones generadas en código Erlang como las generadas en código C para *structs* y *unions* opacos se limitan a llamar a las funciones de tratamiento de punteros. En el caso de los *unions*, ya se trataban exclusivamente como punteros en su versión «no opaca», de modo que no se aprecian diferencias en este aspecto.

Siguiendo con el tipo SDL_Window, las funciones generadas en código Erlang son las siguientes:

```
window_to_bytelist(Value) ->
   pointer_to_bytelist(Value).

bytelist_to_window(Bytelist) ->
   bytelist_to_pointer(Bytelist).

parse_window(Bytelist) ->
   parse_pointer(Bytelist).
```

Y del mismo modo en código C se obtiene lo siguiente:

```
byte * read_window(byte *in, SDL_Window *result) {
   return read_pointer(in, (void **) &result);
}

byte * write_window(SDL_Window *value, byte *out, size_t *len) {
   return write_pointer((void **) &value, out, len);
}
```

Cabe destacar que al no tener acceso a la definición de estos tipos de datos (y por tanto, a sus elementos), en este caso no se generan funciones de tipo *get* y *set*.

Con esto concluye el contenido de este capítulo en el que se ha descrito la generación de *enums*, *structs* y *unions*. La inclusión de este tipo de elementos en la herramienta de generación de código permite generar prácticamente la totalidad de funciones de la librería SDL. En los siguientes capítulos se van a tratar algunos aspectos que quedan pendientes,

como el tratamiento de *arrays* y la generación de algunas funciones auxiliares para facilitar el trabajo del programador a la hora de usar punteros C desde Erlang.

Capítulo 5

Generación de código: punteros

Este capítulo se centra en aspectos relacionados con el tratamiento de punteros a la hora de generar los bindings. A pesar de que en capítulos anteriores ya se ha descrito cómo tratar los punteros de C en Erlang (ver sección 3.3.5), aquí se va hacer hincapié en otras funciones generadas para cada tipo de dato que están relacionadas directamente con estos, como son la creación y destrucción de memoria dinámica desde Erlang y la indirección (también llamada desreferenciación).

5.1. Creación y destrucción de punteros

A la hora de trabajar con los bindings generados, es posible que el programador de Erlang necesite utilizar una variable de tipo puntero que no le sea posible obtener mediante alguna de las funciones de la librería C. En este caso, el programador es el responsable de reservar espacio en la memoria dinámica, para después obtener un puntero a la región de memoria reservada. El hecho de que los punteros manejados desde Erlang son realmente punteros de C, cuya memoria es reservada y gestionada por el programa C, hace que el programador no cuente con ningún mecanismo nativo para crearlos de la misma forma que en C se haría con la función `malloc`. Por otra parte, este mismo hecho imposibilita al programador liberar desde Erlang la memoria de un puntero* que ya no se necesite, ya que el propio recolector

*A no ser que la librería ofrezca algún mecanismo de liberación de memoria propio para algunos tipos de datos, como en el caso de SDL la función `SDL_FreeSurface`, que libera la memoria de una variable de tipo `SDL_Surface`

de basura de Erlang entiende ese tipo de dato como un simple entero de 64 bits.

Como solución a esta limitación la herramienta genera para cada tipo de dato (*int*, *float*, *struct*, *union*, etc.) una función *new* y una función *delete* que se encargan de reservar y liberar memoria desde Erlang en el programa C. De este modo, el programador puede llevar a cabo la gestión de memoria de manera explícita en aquellos casos en los que sea necesario. Estas funciones son generadas en los tipos de datos básicos, así como en los tipos de datos definidos por el programador en el fichero de especificación.

Si analizamos estas funciones desde el punto de vista de Erlang, tenemos que:

- Las funciones `new_XXX` (donde *XXX* es el tipo de dato) se encargan de realizar una petición a C a través del puerto (como si de cualquier otra función se tratase) con el fin de que el propio programa C reserve memoria para un tipo *XXX* y devuelva a Erlang el puntero que contiene esta dirección.

- Las funciones `delete_XXX` realizan una petición a C enviando un puntero como parámetro y el programa C se encarga de liberar el espacio de memoria reservada al que apunta.

La estructura de estas funciones en Erlang es similar en todos los tipos de datos. En la siguiente sección de código puede verse un ejemplo para el tipo `int32`.

```
new_int32() ->
   Code = int_to_bytelist(19),    % <- Código de operación
   ResultCall = call_port_owner(?PORT_NAME, [Code]),
   case ResultCall of
      {datalist, DataList} ->
         bytelist_to_pointer(DataList);
      Msg ->
         {error, Msg}
   end.

delete_int32(Pointer) ->
   Code = int_to_bytelist(21),    % <- Código de operación
   PList = pointer_to_bytelist(Pointer),
   ResultCall = call_port_owner(?PORT_NAME, [Code, PList]),
```

```
   case ResultCall of
      {datalist, _DataList} ->
         ok;
      Msg ->
         {error, Msg}
   end.
```

Por la parte del código C, se generan las funciones correspondientes `new_XXX_Handler` y `delete_XXX_Handler` que se encargan de atender la petición de Erlang reservando y liberando memoria respectivamente como puede verse en el siguiente ejemplo para el tipo `int32`.

```
void new_int32_Handler(byte *in, size_t len_in,
                       byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in += 4;

   int *ptr = malloc(sizeof(int));
   current_out = write_pointer((void **) &ptr, current_out, len_out);
}

void delete_int32_Handler(byte *in, size_t len_in,
                          byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in += 4;

   int *ptr;
   current_in = read_pointer(current_in, (void **) &ptr);
   free(ptr);
}
```

5.2. Indirección de punteros

Una vez resuelta la creación de punteros, es necesario darle al programador un mecanismo para desreferenciar el puntero y poder obtener el valor al que apunta, similar a lo que hace el operador `*` en C. Del mismo modo, también es necesario proporcionar un mecanismo para asignar un valor en las direcciones de memoria apuntadas por los punteros.

Al igual que ocurre con el caso anterior, debido a que en Erlang estos punteros solo son simples enteros carentes de significado y que la región de memoria apuntada por ellos se encuentra en el programa C, es necesaria la creación de nuevas funciones auxiliares para cada tipo de dato que realicen peticiones, para que la parte de C se encargue de estos procedimientos. Estas tareas son realizadas por las funciones `pointer_deref_XXX` y `pointer_deref_XXX_assign`, que al igual que `new_XXX` y `delete_XXX` se generan para cada tipo de dato. El procedimiento de estas funciones es el siguiente:

- `pointer_deref_XXX`: Recibe como parámetro el puntero a desreferenciar y lo serializa junto con el código de operación para enviar la petición al puerto. A continuación permanece a la espera de recibir el dato de tipo XXX en formato lista de bytes y lo deserializa.

- `pointer_deref_XXX_assign`: Recibe como parámetro el puntero y el valor de tipo XXX que va a asignarse a la dirección de memoria apuntada. Serializa estos datos junto con el código de operación y en este caso recibe una respuesta vacía.

Siguiendo con el ejemplo del tipo `int32`, este sería el código generado:

```
pointer_deref_int32(Pointer) ->
   Code = int_to_bytelist(15),    % <- Código de operación
   PList = pointer_to_bytelist(Pointer),
   ResultCall = call_port_owner(?PORT_NAME, [Code, PList]),
   case ResultCall of
      {datalist, DataList} ->
         bytelist_to_int(DataList, 32);
      Msg ->
         {error, Msg}
   end.

pointer_deref_int32_assign(Pointer, Value) ->
   Code = int_to_bytelist(17),    % <- Código de operación
   PList = pointer_to_bytelist(Pointer),
   VList = int_to_bytelist(Value, 32),
   ResultCall = call_port_owner(?PORT_NAME, [Code, PList, VList]),
   case ResultCall of
```

```
      {datalist, _DataList} ->
         ok;
      Msg ->
         {error, Msg}
   end.
```

En C se generan las funciones manejadoras equivalentes para cada tipo:

- `pointer_deref_XXX_Handler`: Lee del *buffer* de entrada el puntero, utiliza el operador `*` para obtener el valor al que apunta y lo devuelve a Erlang a través del *buffer* de salida del puerto.
- `pointer_deref_XXX_assign_Handler`: Lee del *buffer* de entrada el puntero y el valor del tipo XXX para realizar la asignación mediante el operador `*`.

El código generado en C para el tipo `int32` sería el siguiente:

```
void pointer_deref_int32_Handler(byte *in, size_t len_in,
                                 byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in += 4;

   int *ptr;
   current_in = read_pointer(current_in, (void **) &ptr);
   current_out = write_int32(ptr, current_out, len_out);
}

void pointer_deref_int32_assign_Handler(byte *in, size_t len_in,
                                        byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in += 4;

   int *ptr, value;
   current_in = read_pointer(current_in, (void **) &ptr);
   current_in = read_int32(current_in, &value);
   *ptr = value;
}
```

Con el fin de mostrar un ejemplo de uso de las funciones introducidas en este capítulo, en el siguiente listado puede verse un fragmento de la shell de Erlang donde ser crea

un puntero a `int` haciendo uso de la función `new_int`, se le asigna el valor 166 mediante `pointer_deref_int_assign` y se hace una indirección con `pointer_deref_int`[**].

```
9> N1 = sdl_ports_gen:new_int().
140506379084064
10> sdl_ports_gen:pointer_deref_int_assign(N1, 166).
ok
11> sdl_ports_gen:pointer_deref_int(N1).
166
```

Con estas nuevas funciones el programador ahora dispone de nuevos mecanismos para trabajar con punteros, lo cual hace que la programación en Erlang haciendo uso de los *bindings* generados sea más «amigable» y funcional. El siguiente paso en la mejora de la usabilidad del código generado es la manipulación de datos de tipo *array*, que va a ser tratada en el siguiente capítulo.

[**]Estas tres funciones, internamente, se encargan de llamar a las funciones `new_int32`, `pointer_deref_int32_assign` y `pointer_deref_int32` respectivamente, ya que los datos de tipo `int` tienen por defecto un tamaño de 32 bits.

Capítulo 6

Generación de código: *arrays*

Los *arrays* (o vectores) son elementos muy comunes en C y otros lenguajes que se caracterizan por representar una colección de datos del mismo tipo almacenados en memoria de manera contigua [3]. Estos datos son accesibles mediante índices dentro del array. En C podemos encontrar dos tipos de *arrays*:

- Estáticos: El tamaño del *array* se define en tiempo de compilación y no puede ser modificado.
- Dinámicos: El tamaño se define en tiempo de ejecución. El *array* se declara como un puntero para el cual se puede reservar una cantidad de memoria dinámica en cualquier momento de la ejecución.

Ejemplo de declaración de *arrays* en C:

```
int array_estatico[10];
int *array_dinamico = malloc(10 * sizeof(int));
```

Cuando se pasa un *array* como parámetro a una función, se considera que es un *array* dinámico. Aunque la llamada a la función se haga pasándole un *array* estático, lo que la función recibe realmente es un puntero a la dirección de memoria donde se encuentra el primer elemento del array. Por tanto, el array se pasa a la función por referencia, y no por valor. El único modo de pasar por valor un *array* estático como parámetro de una función es mediante su integración como elemento de un *struct*, ya que estos últimos sí se pasan por

valor, realizándose la copia de todos los datos del *struct* al contexto de la función llamada, en lugar de pasar un puntero a esos datos.

El tipo *array* estático tiene su propia descripción de tipo dentro del fichero de especificación, en la cual se indica el átomo identificador `fixed_array`, el tipo de dato y el número de elementos.

En el siguiente ejemplo puede verse la definición en C de un `struct` (`ArrayA`) que contiene un *array* estático y a continuación su representación en el fichero de especificación.

```
typedef struct {
   int id;
   int values[10];
} ArrayA;
```

```
{spec,
   ...
   [
      {type_spec, arrayA, "ArrayA",
         {struct, [
            {struct_member, id, "id", int, []},
            {struct_member, values, "values", {fixed_array, int, 10}, []}
            ]
         }, []},
      ...
   ]
   ...
}.
```

El hecho de que el concepto de *array* como tal no exista en Erlang y de que sea un tipo tan cotidiano en el uso de cualquier librería de C hace necesaria la implementación de mecanismos que permitan manejar este tipo de datos en los *bindings* generados.

Para tratar los *arrays* estáticos en Erlang se ha optado por utilizar la estructura que guarda más similitud con estos: las listas. También se contempla la opción de manejar *arrays* dinámicos almacenados en la parte de C mediante punteros, de forma similar a como se manejan los *union* (ver sección 4.3). En base a estas ideas la herramienta genera una serie de funciones con el fin de facilitar el uso de este tipo de datos en la mayor medida posible.

6.1. Serialización de *arrays*

6.1.1. Generación de código Erlang

Para que en Erlang se pueda trabajar con listas que sean entendidas por el código C como *arrays* es necesario generar, para cada tipo de dato X, nuevas funciones que realicen la conversión de lista de X a lista de bytes y viceversa. Estas funciones existen solamente para uso interno por parte del resto de funciones generadas. No se espera que el programador llame directamente a estas funciones.

Concretamente la herramienta genera, para cada tipo de dato, tres funciones nuevas:

- `XXX_array_to_bytelist(L, S)`: Convierte una lista L que contiene S datos de tipo XXX en una lista de bytes que corresponde a la serialización de cada uno de los S elementos de la lista L.
- `bytelist_to_XXX_array(L, S)`: Convierte una lista L de bytes en una serie de S elementos de tipo XXX y los almacena en una lista Erlang.
- `parse_XXX_array(L, S)`: Similar a la anterior función, pero en este caso toma de lista de bytes L los elementos necesarios para construir la lista de tipo XXX y devuelve ésta en una tupla junto con los bytes sobrantes de la lista L.

La siguiente sección de código ejemplifica la generación de estas funciones para el tipo `SDL_Rect`.

```
rect_array_to_bytelist(List, Size) when length(List) == Size ->
   [ rect_to_bytelist(E) || E <- List].

bytelist_to_rect_array(Bytelist, Size) ->
   bytelist_to_rect_array(Bytelist, Size, []).
bytelist_to_rect_array(_Bytelist, 0, Result) ->
   lists:reverse(Result);
bytelist_to_rect_array(Bytelist, Size, Result) ->
   {Elem, Rest} = parse_rect(Bytelist),
   bytelist_to_rect_array(Rest, Size - 1, [Elem | Result]).
```

```
parse_rect_array(Bytelist, Size) ->
   parse_rect_array(Bytelist, Size, []).
parse_rect_array(Bytelist, 0, Result) ->
   {lists:reverse(Result), Bytelist};
parse_rect_array(Bytelist, Size, Result) ->
   {Elem, Rest} = parse_rect(Bytelist),
   parse_rect_array(Rest, Size - 1, [Elem | Result]).
```

6.1.2. Generación de código C

Del mismo modo es necesario generar en C nuevas funciones para cada tipo de dato X que se encarguen de leer las secuencias de bytes para convertirlas en *arrays* de tipo X y viceversa.

En este caso se generan, para cada tipo, dos funciones:

- `read_XXX_array`: Se encarga de leer del *buffer* de entrada n elementos de tipo XXX y los va almacenando en un *array*.
- `write_XXX_array`: Se encarga de escribir en el *buffer* de salida los n elementos de tipo XXX de un *array*.

Nuevamente tomamos como ejemplo el tipo `SDL_Rect` para plasmar la generación de este tipo de funciones.

```
byte * read_rect_array(byte *in, SDL_Rect *array, int n) {
   byte *current_in = in;

   for (int i=0; i<n; i++)
      current_in = read_rect(current_in, &array[i]);

   return current_in;
}

byte * write_rect_array(SDL_Rect *array, byte *out, size_t *len, int n) {
   byte *current_out = out;

   for (int i=0; i<n; i++)
      current_out = write_rect(&array[i], current_out, len);
```

```
    return current_out;
}
```

6.2. Creación de *arrays* dinámicos y acceso a sus elementos mediante punteros

6.2.1. Generación de código Erlang

Para el manejo de *arrays* dinámicos se va a trabajar en Erlang directamente con punteros. Esto da lugar a que, además de las funciones para punteros explicadas en el capítulo 5, se necesiten nuevos procedimientos que tengan en cuenta la reserva de memoria para este tipo de datos así como el acceso a ella.

A continuación se listan las funciones generadas en Erlang para la creación de *arrays* dinámicos y el acceso a sus elementos:

- `new_XXX_array`: Similar a las funciones `new_XXX` de la sección 5.1, indicándole a C en este caso el número de elementos de tipo *XXX* a reservar en memoria. Devuelve el puntero al *array* creado.
- `pointer_deref_XXX_array`: Similar a las funciones `pointer_deref_XXX` de la sección 5.2, indicando en este caso, además del puntero al *array*, el índice i del dato a obtener. Devuelve el dato de tipo *XXX* que se encuentra en la posición i-ésima del *array*.
- `pointer_deref_XXX_array_assign`: Similar a las funciones `pointer_deref_XXX_assign` de la sección 5.2, indicando en este caso, además del puntero al *array* y el valor a asignar, el índice i del dato a modificar. Esta función devuelve el átomo `ok` en caso de éxito.

Para el tipo `SDL_Rect`, éstas son las funciones generadas por la herramienta:

```
new_rect_array(Size) ->
    Code = int_to_bytelist(170),   % <- Código de operación
```

```
   SList = int_to_bytelist(Size),
   ResultCall = call_port_owner(?PORT_NAME, [Code, SList]),
   case ResultCall of
      {datalist, DataList} ->
         bytelist_to_pointer(DataList);
      Msg ->
         {error, Msg}
   end.

pointer_deref_rect_array(Pointer, Index) ->
   Code = int_to_bytelist(166),  % <- Código de operación
   PList = pointer_to_bytelist(Pointer),
   IList = int_to_bytelist(Index),
   ResultCall = call_port_owner(?PORT_NAME, [Code, PList, IList]),
   case ResultCall of
      {datalist, DataList} ->
         bytelist_to_rect(DataList);
      Msg ->
         {error, Msg}
   end.

pointer_deref_rect_array_assign(Pointer, Index, Value) ->
   Code = int_to_bytelist(168),  % <- Código de operación
   PList = pointer_to_bytelist(Pointer),
   IList = int_to_bytelist(Index),
   VList = rect_to_bytelist(Value),
   ResultCall = call_port_owner(?PORT_NAME, [Code, PList, IList, VList]),
   case ResultCall of
      {datalist, _DataList} ->
         ok;
      Msg ->
         {error, Msg}
   end.
```

6.2.2. Generación de código C

Las funciones generadas en C, al igual que en secciones anteriores, son los *handlers* que dan respuesta a las peticiones realizadas por las funciones de Erlang.

- `new_XXX_array_Handler`: Actúa de la misma forma que `new_XXX_Handler` de la sección

5.1, pero en ese caso lee el tamaño del *array* y reserva memoria para ese número de datos de tipo *XXX* antes de devolver a Erlang el puntero.

- `pointer_deref_XXX_array_Handler`: Actúa de la misma forma que `pointer_deref_XXX_Handler` (sec. 5.2), pero en este caso se lee el puntero y un índice, devolviendo a Erlang el valor de tipo *XXX* que corresponde a ese indice del *array*.

- `pointer_deref_XXX_array_assign_Handler`: Actúa de la misma forma que `pointer_deref_XXX_assign_Handler` (sec. 5.2), pero en este caso se lee el puntero, un valor de tipo *XXX*, y un índice, modificando el contenido de dicho indice del *array* con el valor leído.

De nuevo, para mostrar un ejemplo de este tipo de funciones se ha tomado el tipo `SDL_Rect`.

```
// Manejador para la función de creación de arrays
void new_rect_array_Handler(byte *in, size_t len_in,
                            byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in += 4;

   int size;
   // Obtenemos tamaño del array a crear
   current_in = read_int(current_in, &size);
   // Reservamos memoria para el array
   SDL_Rect *ptr = malloc(sizeof(SDL_Rect)*size);
   // Devolvemos el puntero al array recién creado
   current_out = write_pointer((void **) &ptr, current_out, len_out);
}

// Manejador para la función de acceso a un array
void pointer_deref_rect_array_Handler(byte *in, size_t len_in,
                                      byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in += 4;

   SDL_Rect *ptr;
   int index;
   // Obtenemos puntero al array y posición a leer
```

```
   current_in = read_pointer(current_in, (void **) &ptr);
   current_in = read_int(current_in, &index);
   // Devolvemos la posición leída (ptr[index])
   current_out = write_rect(&(ptr[index]), current_out, len_out);
}

// Manejador para la función de modificación de un array
void pointer_deref_rect_array_assign_Handler(byte *in, size_t len_in,
                                             byte *out, size_t *len_out) {
   byte *current_in = in, *current_out = out;
   *len_out = 0; current_in+=4;

   SDL_Rect *ptr, value;
   int index;
   // Obtenemos puntero al array, posición a escribir, y valor a escribir.
   current_in = read_pointer(current_in, (void **) &ptr);
   current_in = read_int(current_in, &index);
   current_in = read_rect(current_in, &value);
   // Escribimos en la posición indicada.
   ptr[index] = value;
}
```

6.3. Conversión directa entre *arrays* C y listas Erlang

La interfaz proporcionada por las funciones descritas en los apartados anteriores está en correspondencia directa con las funciones y operadores de gestión de arrays en C, las cuales son de bajo nivel. Esto hace que la gestión de arrays dinámicos por parte del programador que utilice estos *bindings* sea algo tediosa y bastante alejada del modelo de programación habitual en Erlang.

Para facilitar la tarea del programador que haga uso de estos *bindings*, se ha pensado en la generación automática de unas funciones extra en el código Erlang que, haciendo uso de las mencionadas anteriormente en este mismo capítulo, realizan una conversión directa de listas Erlang a *arrays* C, así como la operación contraria.

Dichas funciones actúan de la siguiente manera:

- `list_to_XXX_array`: Recibe una lista con elementos de tipo *XXX* y se encarga de

solicitar a C la creación del *array* mediante `new_XXX_array` y el tamaño de la lista, seguido de sucesivas llamadas a `pointer_deref_XXX_array_assign` para ir rellenando las distintas posiciones del *array* con los valores de la lista. Finalmente devuelve el puntero al *array* creado.

- `XXX_array_to_list`: En este caso se recibe el puntero al *array*, así como el tamaño del mismo. Para ello se realizan diversas llamadas a la función `pointer_deref_XXX_array` para ir obteniendo los distintos valores que componen el *array* y almacenándolos en una lista que es devuelta al final.

Para la realización de estas funciones de alto nivel no es necesaria la generación de código C auxiliar. A modo de ejemplo, estas funciones para `SDL_Rect` se generan de la siguiente manera:

```
list_to_rect_array(List) ->
   Size = length(List),
   Pointer = new_rect_array(Size),
   list_to_rect_array(List, Pointer, 0).
list_to_rect_array([], Pointer, _Index) -> Pointer;
list_to_rect_array([Value|List], Pointer, Index) ->
   pointer_deref_rect_array_assign(Pointer, Index, Value),
   list_to_rect_array(List, Pointer, Index + 1).

rect_array_to_list(Pointer, Size) ->
   rect_array_to_list(Pointer, Size - 1, []).
rect_array_to_list(_Pointer, Size, Result) when Size < 0 -> Result;
rect_array_to_list(Pointer, Size, Result) ->
   Elem = pointer_deref_rect_array(Pointer, Size),
   rect_array_to_list(Pointer, Size - 1, [Elem|Result]).
```

6.4. Mejora de *structs* con *arrays*

La introducción de las funciones de manejo de *arrays* abren la posibilidad de introducir mejoras como la que se desarrolla en esta sección.

Supongamos que tenemos el siguiente `struct` definido en la librería de C que estamos adaptando:

```
typedef struct {
   int id;
   int *values;
   int size;
} ArrayC;
```

Esta estructura cuenta entre sus elementos con un puntero a `int` (*values*) que representa un *array* dinámico, así como una variable int (*size*) que representa el tamaño de dicho *array*. Este tipo de estructura que incluye un *array* en un campo y su tamaño en otro campo es un patrón bastante común en C. Por este motivo se ha incluido una opción en el fichero de especificación que permite al programador reflejar esta situación. La opción en cuestión se ha denominado `{length_of, NumMember}` e indica que un elemento de un `struct` representa el tamaño de otro de sus elementos, concretamente, el que se encuentra en la posición *NumMember* (comenzando a contar desde 1), siempre que este sea un *array* dinámico (esto es, un puntero).

Para el ejemplo de la estructura anterior, el fichero de especificación quedaría de la siguiente manera:

```
{type_spec, arrayC, "ArrayC",
   {struct, [
      {struct_member, id, "id", int, []},
      {struct_member, values, "values", {pointer, int}, []},
      {struct_member, size, "size", int, [{length_of, 2}]}
      ]}, []}
```

El objetivo de contemplar este tipo de patrón en el fichero de especificación es que la herramienta sea capaz de generar para esas estructuras una versión especial de la función *get* que permita obtener el elemento *array* como una lista Erlang en lugar de como un puntero (y de igual forma con la función *set*), todo esto haciendo uso de las funciones ya generadas para *structs* y *arrays*.

Dichas funciones son las siguientes:

- `XXX_get_arraylist_YYY`: Esta función a partir del puntero a la estructura *XXX* obtiene su elemento *YYY*, que contiene el puntero al *array* de tipo *ZZZ*, así como el elemento que contiene su tamaño, y se los pasa a la función `ZZZ_array_to_list` (ver sección 6.3) para que construya la lista Erlang con los elementos del *array*.

- `XXX_set_arraylist_YYY`: Esta función obtiene de igual forma el puntero al elemento *array* de tipo *ZZZ* y su tamaño, para posteriormente realizar una serie de llamadas a `pointer_deref_ZZZ_array_assign` (sec. 6.2) e ir rellenando el *array* con los valores de una lista Erlang proporcionada.

El código generado para el ejemplo de la estructura `ArrayC` sería el siguiente:

```
arrayC_get_arraylist_values(Pointer) ->
   ArrayPtr = arrayC_get_values(Pointer),
   Size = arrayC_get_size(Pointer),
   int_array_to_list(ArrayPtr, Size).

arrayC_set_arraylist_values(Pointer, List) ->
   ArrayPtr = arrayC_get_values(Pointer),
   Size = arrayC_get_size(Pointer),
   arrayC_set_arraylist_values(ArrayPtr, List, Size, 0).
arrayC_set_arraylist_values(_ArrayPtr, _List, Size, Index) when Size == Index
    -> ok;
arrayC_set_arraylist_values(ArrayPtr, [Value|List], Size, Index) ->
   pointer_deref_int_array_assign(ArrayPtr, Index, Value),
   arrayC_set_arraylist_values(ArrayPtr, List, Size, Index+1).
```

6.5. Mejora de funciones con *arrays*

También es posible introducir ciertas mejoras en las llamadas a funciones mediante el uso de los mecanismos para manejar *arrays* que han sido expuestos en este capítulo.

Para explicar el objetivo de la mejora que se va a introducir vamos a tomar como referencia la siguiente función `max`, suponiendo que esta perteneciera a las funciones definidas en la librería C para cual se esta aplicando la herramienta. Esta función recibe un puntero

a un *array* de tipo `int`, así como su tamaño, y se encarga de calcular el entero de mayor valor dentro del array.

```
int max (int *array, int size) {
   int max = INT_MIN;

   for (int i = 0; i < size; i++) {
      if (array[i] > max) max = array[i];
   }

   return max;
}
```

Al igual que ocurre con el caso del *struct* de la sección 6.4, en C es bastante común que aquellas funciones que reciban un array como parámetro tengan que recibir un parámetro adicional indicando la longitud del array. Por este motivo, se ha incluido en el fichero de especificación la opción `{length_of, NumParam}` para indicar que un parámetro de una función representa el tamaño de otro parámetro (*array*) identificado por *NumParam*.

La función `max` en le fichero de especificación quedaría de la siguiente manera:

```
{fun_spec, maxint, "max", [
   {param_spec, {pointer, int}, []},
   {param_spec, int, [{length_of, 1}, free_after]}
   ], int, []},
```

La mejora que permite introducir esto consiste en generar, aparte de la función `max` (llamada `maxint` en Erlang) como se generaba hasta el momento, una nueva versión de esta en la que, en lugar de pasarle como parámetro el puntero al *array* y el tamaño, se le pase simplemente una lista (de enteros en este caso). De este modo, a partir de una lista de entrada, esta función se encargará de:

1. Crear el *array* en C, asignarle los valores de la lista y obtener su puntero mediante la función `list_to_int_array`.

2. Llamar a la función `maxint` original con el puntero al *array* y el tamaño de la lista y obtener el resultado.

3. En el caso en que se haya introducido la opción `free_after` entre las opciones del parámetro dentro del fichero de especificación, liberar la memoria del *array* en C mediante el uso de la función `delete_int`.

La siguiente sección de código muestra tanto la función `maxint` original como la nueva versión generada que recibe una lista:

```
maxint(P_Int_1, Int_2) ->
   Code = int_to_bytelist(770),
   Param1 = pointer_to_bytelist(P_Int_1),
   Param2 = int_to_bytelist(Int_2),

   ResultCall = call_port_owner(?PORT_NAME, [Code, Param1, Param2], []),
   case ResultCall of
      {datalist, DataList} ->
         {RetParamAux, _R1} = parse_int(DataList),
         RetParam1 = RetParamAux,
         RetParam1;
      Msg ->
         {error, Msg}
   end.

maxint(List1) ->
   P_Int_1 = list_to_int_array(List1),
   Int_2 = length(List1),
   Value = maxint(P_Int_1, Int_2),
   delete_int(P_Int_1),
   Value.
```

Adicionalmente, se ha introducido otra opción más en el fichero de especificación para los parámetros de función que tienen la opción `length_of`. Esta opción es `free_after`, la cual indica al generador de código si esta segunda versión de la función debe liberar la memoria del *array* en C una vez haya terminado su operación. En el ejemplo anterior puede verse que se hace uso de esta opción, ya que tras la llamada a `maxint` no se va hacer uso del array intermedio. Sin embargo, hay ocasiones en las que una función de una librería C recibe un array para guardar una referencia al mismo (en una variable global) con el fin de usarla posteriormente. En este caso, sería incorrecto liberar el array tras la llamada a la función.

Como se ha hecho en los capítulos anteriores, a continuación se muestra un ejemplo de uso de de las nuevas funciones introducidas. En este ejemplo se realiza una conversión directa entre una lista de enteros de Erlang y un *array* de C (y viceversa) y, seguidamente, se llama a la función `maxint` con la lista Erlang inicialmente declarada.

```
14> List = [1,2,3,4,5,6,7,8,9,10].
[1,2,3,4,5,6,7,8,9,10]
15> ArrayPtr = sdl_ports_gen:list_to_int_array(List).
140506380108960
16> sdl_ports_gen:int_array_to_list(ArrayPtr, length(List)).
[1,2,3,4,5,6,7,8,9,10]
17> sdl_ports_gen:maxint(List).
10
```

Tras introducir en este capítulo las funciones de manejo de *arrays*, se ha mejorado un poco más la usabilidad de los *bindings* generados. No obstante, aún se pueden introducir algunas mejoras más para aumentar la funcionalidad del código generado. Se trata de la implementación de un sistema de liberación automática de memoria dinámica para los punteros de C desde Erlang y la compatibilidad con funciones de orden superior. Ambos temas serán introducidos en los próximos capítulos.

Capítulo 7

Recolección de basura

Como ya se ha comentado en capítulos anteriores, la gestión de punteros en Erlang se realiza mediante distintas funciones que envían peticiones al programa C para que este encargue de tareas como reserva de memoria, liberación, o indirección, manteniendo en Erlang solo el entero que representa la dirección de memoria. El problema surge cuando el programador no libera manualmente un puntero y entra en juego el recolector de basura de Erlang, ya que cuando este libera la variable que contiene el puntero C solo está liberando el entero que contiene dirección de memoria, quedando en el programa C la sección de memoria referenciada por el puntero, sin posibilidad de ser liberada ni de acceder a ella.

Continuando con la introducción de funcionalidades adicionales y mejoras en la generación de código, este capítulo va a abordar la inclusión de un mecanismo que permita hacer uso del recolector de basura de Erlang para liberar regiones de memoria de C, de modo que el programador no tenga que preocuparse de la liberación manual de memoria cuando maneja punteros C desde Erlang.

7.1. Punteros *raw* y *managed*

Hasta el momento, el tratamiento de punteros en Erlang se ha basado en almacenar su dirección de memoria en forma de número entero. Sin embargo, para poder dar mas versatilidad al código generado y poder aprovecharse del recolector de basura de Erlang, es necesario distinguir entre dos tipos de punteros:

- *Raw pointers*: Este tipo de puntero es similar al que se estaba utilizando hasta el momento, con la diferencia de que ahora la dirección se encuentra dentro de una tupla, con el fin de poder realizar distinción entre los dos tipos de puntero. Esta tupla contiene dos elementos: el átomo `raw_pointer` y el número entero de 64 bits que representa la dirección de memoria (`{raw_pointer, RawPtr}`).

- *Managed pointers*: La representación de este tipo de punteros es un poco más compleja ya que, a pesar de que también se representan mediante una tupla de dos elementos, esta no contienen explícitamente la dirección de memoria del puntero, sino que almacena la referencia a una estructura de datos (de tipo `pointer_wrapper`) en un módulo NIF independiente que hace el papel de intermediario para la gestión de este tipo de punteros. En este caso la estructura de la tupla en Erlang consiste en el átomo `managed_pointer` seguido de la referencia a la estructura `pointer_wrapper` mencionada anteriormente (`{managed_pointer, RefPtrWrap}`).

Una estructura `pointer_wrapper` es totalmente opaca desde el punto de vista del programador, pero actúa como envoltorio de un puntero C. La idea clave reside en el hecho de que Erlang permite asociar a estas estructuras una función *callback* destructora. Esta función es llamada cuando el recolector de basura libera la región de memoria ocupada por la misma. De este modo, cuando el recolector de basura libera un `pointer_wrapper`, la función *callback* debe encargarse de acceder al puntero envuelto y liberar la memoria referenciada por este. Suponemos que, para cada puntero de tipo *managed* que queramos mantener, existe una única estructura `pointer_wrapper` que lo envuelve, independientemente de que el programador Erlang, en su código, pueda tener varias referencias a este `pointer_wrapper`. Cuando desaparezcan todas las referencias a un `pointer_wrapper`, el recolector de basura procederá a eliminarlo, lo cual provocará la llamada a la función destructora, que liberará la región de memoria apuntada por el puntero envuelto.

La estructura `pointer_wrapper` se encuentra definida dentro de una librería en C llamada *erlang_gc*. Pese a que se encuentra implementada en lenguaje C, no debe confundirse

con el código C generado a partir del fichero de especificación. El módulo *erlang_gc* define la estructura `pointer_wrapper` e implementa una serie de NIFs de acceso esta estructura. Al ser NIFs, estas funciones son llamadas directamente por la máquina virtual de Erlang. Por el contrario, el código C generado por nuestra herramienta está pensado para ser ejecutado como un proceso aparte del sistema operativo (mediante la función `open_port`), cuya comunicación con la máquina virtual de Erlang se realiza mediante puertos.

A continuación se muestra la definición de `pointer_wrapper`:

```
typedef struct {
   // Nombre del módulo que contiene la función destructora
   char *free_module_name;
   // Nombre de la función destructora
   char *free_function_name;
   // Puntero a destruir cuando le toque recolección de basura
   uintptr_t wrapped_ptr;
} pointer_wrapper;
```

Debido a la diferenciación entre estos dos nuevos tipos de punteros en Erlang es necesaria la generación de una nueva función que se encargue de obtener el entero con la dirección de memoria del puntero para que esta pueda ser enviada a través del puerto y utilizada por el programa C.

- En el caso de recibir un *raw pointer*, esta función se limita a revolver la dirección de memoria almacenada en la segunda componente de la tupla.
- Si, en cambio, el puntero recibido es de tipo *managed*, se encarga de realizar una llamada a la función NIF `get_wrapped_pointer` del módulo *erlang_gc* que obtiene la dirección de memoria del puntero haciendo uso del campo `wrapped_ptr` de la estructura `pointer_wrapper` correspondiente.

En el siguiente listado puede verse la implementación de esta función para cada tipo de puntero.

```
get_ptr({raw_pointer, Ptr}) -> Ptr;
get_ptr({managed_pointer, _} = Managed) ->
```

```
erlang_gc:get_wrapped_pointer(Managed).
```

Además de la inclusión de esta nueva función, es necesario introducir ciertos cambios en algunas de las funciones que hasta ahora se encargaban de tareas como la serialización de punteros, de modo que la utilización de estos nuevos tipos no afecte a las funciones de comunicación con C generadas hasta el momento.

Podemos destacar los siguientes cambios:

- `pointer_to_bytelist`: Se encarga obtener el entero con la dirección de memoria del puntero mediante la función `get_ptr` y una vez obtenido realizar la conversión a lista de bytes.
- `bytelist_to_pointer`: Convierte la lista de bytes en el entero que representa la dirección de memoria del puntero y construye una tupla de tipo *raw pointer*.
- `parse_pointer`: Actúa de la misma manera que la anterior pero además devuelve el resto de la lista de bytes recibida.

El siguiente listado de código muestra la nueva implementación de estas tres funciones.

```
pointer_to_bytelist(Value) ->
   int_to_bytelist(get_ptr(Value), 64).

bytelist_to_pointer(Bytelist) ->
   {raw_pointer, bytelist_to_int(Bytelist, 64)}.

parse_pointer(Bytelist) ->
   {RawPtr, RestBytes} = parse_int(Bytelist, 64),
   {{raw_pointer, RawPtr}, RestBytes}.
```

7.2. Módulo de gestión de punteros *managed*

El módulo de gestión de punteros de tipo *managed*, denominado *erlang_gc*, es el encargado de gestionar la memoria de los punteros C que son manipulados desde Erlang. En otras

palabras, este es el módulo que se encarga de gestionar las estructuras `pointer_wrapper` y de la llamada a las funciones destructoras correspondientes en el caso en que las primeras sean liberadas por el recolector de basura de Erlang.

La descripción del funcionamiento de este módulo comienza por su inicialización. Esta tiene lugar durante la inicialización del puerto Erlang, en módulo principal de los *bindings* generados. La llamada a la función `code:ensure_loaded(erlang_gc)` se encarga de cargar el módulo *erlang_gc* que, al inicializarse, ejecuta su función `init()`, la cual carga la librería de C que contiene las NIFs y lanza un proceso registrado (*memory_manager*) para recibir peticiones de liberación de punteros.

En el siguiente listado de código se presenta la inicialización del módulo *erlang_gc* dentro de la inicialización del puerto de la librería SDL, el caso de estudio de este trabajo.

```
% Fichero sdl_ports_gen.erl
init_port() ->
   Pid = spawn(fun() -> start_port_owner("sdl_ports_gen") end),
   register(?PORT_NAME, Pid),
   % Cargamos módulo erlang_gc, en el caso en que no estuviera
   % cargado ya.
   code:ensure_loaded(erlang_gc),
   io:format("sdl_port initialized.~n"),
   ok.

% Fichero erlang_gc.erl
init() ->
   erlang:load_nif("./gc_nif", 0),
   MemManager = spawn(fun() -> memory_manager_loop() end),
   register(memory_manager, MemManager),
   io:format("Memory manager initialized~n"),
   ok.
```

Una vez lanzado el proceso *memory_manager*, este se ejecuta en bucle de manera continua para recibir peticiones de liberación de memoria. Estas peticiones tienen la estructura `{free, ModuleName, FunctionName, Ptr}`, donde:

- `free`: Es el átomo identificador de la petición.

- `ModuleName`: Es el nombre del módulo donde se encuentra la función de liberación de memoria.
- `FunctionName`: Es el nombre de la función de liberación de memoria.
- `Ptr`: La dirección de memoria a liberar.

Cuando se recibe la petición, el proceso se encarga de llamar a la función indicada en ella, que será la que se encargue de liberar la memoria del puntero `Ptr`. El código que ejecuta el proceso *memory_manager* puede verse en el listado mostrado a continuación:

```
memory_manager_loop() ->
   receive
      { free, ModuleName, FunctionName, Ptr } ->
         ModuleName:FunctionName({raw_pointer, Ptr}),
         memory_manager_loop()
   end.
```

Por último, queda describir las funciones NIFs que pertenecen a este módulo. Estas son `manage_ptr` y `get_wrapped_pointer` y se describen a continuación:

- `manage_ptr`: Esta función es llamada cuando se pretende que un *raw pointer* pase a ser *managed pointer*, y por tanto su memoria sea liberada por el recolector de basura. Recibe el nombre del módulo y de la función destructora, así como la tupla *raw pointer*. Se encarga de almacenar en C estos datos en la estructura `pointer_wrapper` (ver sección 7.1), construir la tupla *managed pointer* con la referencia a esta estructura y la devolverla al proceso Erlang.
- `get_wrapped_pointer`: Esta es la función a la que se llama cuando se quiere obtener la dirección de memoria de un *managed pointer* mediante `get_ptr` (ver sección 7.1). Recibe una tupla *managed pointer* y, a través de la referencia a la estructura `pointer_wrapper`, obtiene la dirección del puntero en forma de número entero y la devuelve al proceso Erlang.

Para que se produzca la liberación de memoria de los *managed pointer* de forma automática se define en la librería C del módulo NIF *erlang_gc* una función destructora `pointer_wrapper_destructor` que es ejecutada cuando el recolector de basura de Erlang elimina la estructura `pointer_wrapper` que almacena todos los datos del puntero. Esta función es la que se encarga de enviar la petición de liberación al proceso *memory_manager* con los datos del puntero a liberar y su función destructora para que se proceda a su liberación.

El código de la función `pointer_wrapper_destructor` es el siguiente:

```
void pointer_wrapper_destructor(ErlNifEnv* env, void* obj) {
   pointer_wrapper *pw = (pointer_wrapper *) obj;

   ErlNifEnv *dst_env = enif_alloc_env();

   // Construimos la tupla { free, ModuleName, FunName, Pointer } ...
   ERL_NIF_TERM result = enif_make_tuple4(dst_env,
      enif_make_atom(dst_env, "free"),
      enif_make_atom(dst_env, pw->free_module_name),
      enif_make_atom(dst_env, pw->free_function_name),
      enif_make_long(dst_env, pw->wrapped_ptr)
   );

   // ...y se la enviamos al proceso que esté registrado como memory manager
   ErlNifPid memory_handler_pid;

   if (enif_whereis_pid(env,
         enif_make_atom(env, MEMORY_MANAGER_NAME), &memory_handler_pid)) {
      // Si existe un proceso registrado con el nombre MEMORY_MANAGER_NAME,
      // se envía la tupla anterior a dicho proceso.
      enif_send(env, &memory_handler_pid, dst_env, result);
   }
}
```

7.3. Otros cambios en la generación de código

Para hacer uso de este sistema de recolección de basura se han implementado una serie de cambios en la generación de código.

El primero de estos cambios afecta al fichero de especificación. Se ha creado una nueva

opción `auto_managed` a incluir en la especificación de funciones que devuelven punteros, cuya finalidad es indicarle al generador de código que la destrucción del puntero devuelto será gestionada por el módulo *erlang_gc* de recolección de basura.

De este modo, si queremos que el puntero de tipo `window` devuelto por la función `create_window` sea liberado automáticamente por el recolector de basura, la especificación queda de la siguiente manera:

```
{fun_spec, create_window, "SDL_CreateWindow", [
   {param_spec, string, []},
   {param_spec, int, []},
   {param_spec, int, []},
   {param_spec, int, []},
   {param_spec, int, []},
   {param_spec, uint32, []}],
   {pointer, window}, [auto_managed]},
```

Cuando el generador de código se encarga de generar una función que contiene esta opción, añade unas líneas adicionales que se encargan de convertir el puntero (entero de 64 bits) devuelto por C en un *managed pointer* que será gestionado por el módulo *erlang_gc*. Esto se hace mediante la función `manage_ptr`, indicándole la función destructora para el tipo de dato al que apunta el puntero y el módulo en el que se encuentra.

Teniendo esto en cuenta, el código resultante para la función `create_window` en Erlang es el siguiente:

```
create_window(String_1, Int_2, Int_3, Int_4, Int_5, Uint32_6) ->
   Code = int_to_bytelist(759),
   Param1 = string_to_bytelist(String_1),
   Param2 = int_to_bytelist(Int_2),
   Param3 = int_to_bytelist(Int_3),
   Param4 = int_to_bytelist(Int_4),
   Param5 = int_to_bytelist(Int_5),
   Param6 = uint32_to_bytelist(Uint32_6),

   ResultCall = call_port_owner(?PORT_NAME, [Code, Param1, Param2, Param3,
   Param4, Param5, Param6], []),
   case ResultCall of
      {datalist, DataList} ->
         {RetParamAux, _R1} = parse_pointer(DataList),
```

```
        case RetParamAux of
           {raw_pointer, P} ->
              % Una vez se recibe el raw pointer, se llama al módulo
              % erlang_gc para transformarlo en managed pointer.
              RetParam1 = erlang_gc:manage_ptr(?MODULE, delete_window, P);
           Error -> RetParam1 = Error
        end,
        RetParam1;
     Msg ->
        {error, Msg}
  end.
```

En el caso de que un tipo de dato concreto cuente con una función específica ofrecida por la librería C para liberar memoria, se cuenta con otra opción adicional para el fichero de especificación que permite indicar el nombre de la misma para que pueda ser incluida en la llamada a la función `manage_ptr`. Esta opción se indica como una tupla `{destructor, NombreFuncion}`, donde *NombreFuncion* es el nombre (en Erlang) de la función destructora de ese tipo de dato.

Por ejemplo, la librería SDL cuenta con una función específica para liberar la memoria de los datos de tipo `SDL_Surface` denominada `SDL_FreeSurface` (cuyo nombre en Erlang es `free_surface`), de modo que su especificación quedaría finalmente de la siguiente manera:

```
{type_spec, surface, "SDL_Surface",
   {struct, [
      {struct_member, flags, "flags", uint32, [internal]},
      {struct_member, format, "format", {pointer,pixel_format}, [read_only]},
      {struct_member, w, "w", int, [read_only]},
      {struct_member, h, "h", int, [read_only]},
      {struct_member, pitch, "pitch", int, [read_only]},
      {struct_member, pixels, "pixels", pointer, []},
      {struct_member, userdata, "userdata", pointer, []},
      {struct_member, locked, "locked", int, [internal]},
      {struct_member, lock_data, "lock_data", pointer, [internal]},
      {struct_member, clip_rect, "clip_rect", rect, [read_only]},
      {struct_member, map, "map", {pointer, blit_map}, [internal]},
      {struct_member, refcount, "refcount", int, []}
      ]
   },
   [{destructor, free_surface}]},
```

Por último, se ha añadido a la generación de código para cada tipo de dato una nueva versión adicional de la función *new_XXX* denominada *new_XXX_auto*, que permite (al igual que su homónima) la creación de un puntero en C para el tipo de dato *XXX*, con la diferencia de que su liberación será gestionada de manera automática por el recolector de basura de Erlang (haciendo uso del módulo *erlang_gc*).

A modo de ejemplo se muestra la implementación de la función `new_int32_auto`, que crea un puntero a entero de 32 bits manejado por el recolector de basura.

```
new_int32_auto() ->
   Pointer = new_int32(),
   case Pointer of
      {raw_pointer, P} ->
         erlang_gc:manage_ptr(?MODULE, delete_int32, P);
      Error -> Error
   end.
```

Una vez introducidas estas mejoras para la liberación de memoria dinámica de punteros C desde Erlang, sólo queda hablar de la última de las funcionalidades a incluir en el generador de código, dentro de la planificación de este proyecto. El siguiente capítulo describe la generación de funciones de orden superior, dando paso finalmente a las conclusiones de este trabajo.

Capítulo 8

Orden superior

En este capítulo se va a introducir la última de las funcionalidades que han sido introducidas en el desarrollo de este trabajo. Consiste en dotar al código generado con la posibilidad de trabajar con funciones de orden superior, en otras palabras, la posibilidad de generar *bindings* de funciones que pueden recibir como parámetro a otras funciones.

8.1. Introducción a las funciones de orden superior y limitaciones

Una función se dice que es de orden superior cuando recibe otra función (o funciones), como parámetro, o devuelve una nueva función como resultado de su ejecución. Muchos lenguajes de programación ofrecen la posibilidad de trabajar con este tipo de funciones, entre ellos los lenguajes de los que estamos haciendo uso en este trabajo: Erlang y C.

A modo de ejemplo para desarrollar durante este capítulo se va a definir la función `apply_int`, que recibe como parámetros un entero `x` y dos funciones `f1` y `f2`. Esta función se encarga de aplicar a `x` las funciones `f1` y `f2` y devolver la suma de sus resultados.

El siguiente listado muestra la implementación de esta función en código C.

```
int apply_int(int x, int (*f1)(int), int (*f2)(int)) {
   return (*f1)(x) + (*f2)(x);
}
```

Las mejoras que se describen en este capítulo permiten que el generador de código

soporte funciones en C de este tipo y obtener sus *bindings* correspondientes. De este modo, el programador de Erlang podría hacer uso de ellas pasando como parámetro λ-abstracciones o referencias a funciones Erlang allá donde se requiera un argumento de tipo funcional.

A continuación, puede verse el uso de este tipo de funciones en un ejemplo de la shell de Erlang donde se llama a la función `apply_int`:

```
6> higher_order:apply_int(3, fun(X) -> X * 2 end, fun(X) -> X + 1 end).
10
```

No obstante, el método empleado (que es descrito en apartados posteriores) presenta algunas limitaciones que cabe destacar:

- Es posible pasar una función como parámetro siempre y cuando esta no reciba a su vez otras funciones como parámetro, ya que esto elevaría considerablemente la complejidad del código, así como el protocolo de comunicación entre Erlang y C.
- Del mismo modo, no se permiten funciones que devuelvan como resultado otras funciones.

El soporte de orden superior en toda su generalidad queda pendiente para trabajo futuro.

8.2. Actualización del sistema de comunicación Erlang-C

La implementación de funciones de orden superior en el generador de código implica una serie de cambios en la generación de código. Entre estos cambios, el que más destaca sin duda es la necesidad de actualizar el sistema de comunicación entre Erlang y C, ya que afecta a todo el código generado hasta el momento.

8.2.1. Actualización del esquema de comunicación

Hasta este momento, el esquema de comunicación entre Erlang y C se venía definiendo de la siguiente manera (ver figura 8.1):

1. Desde Erlang se envía a C una petición de función con el código de operación seguido de los parámetros.

2. Desde C se recibe esta petición, se ejecuta la función identificada por el código y se devuelve el resultado a Erlang.

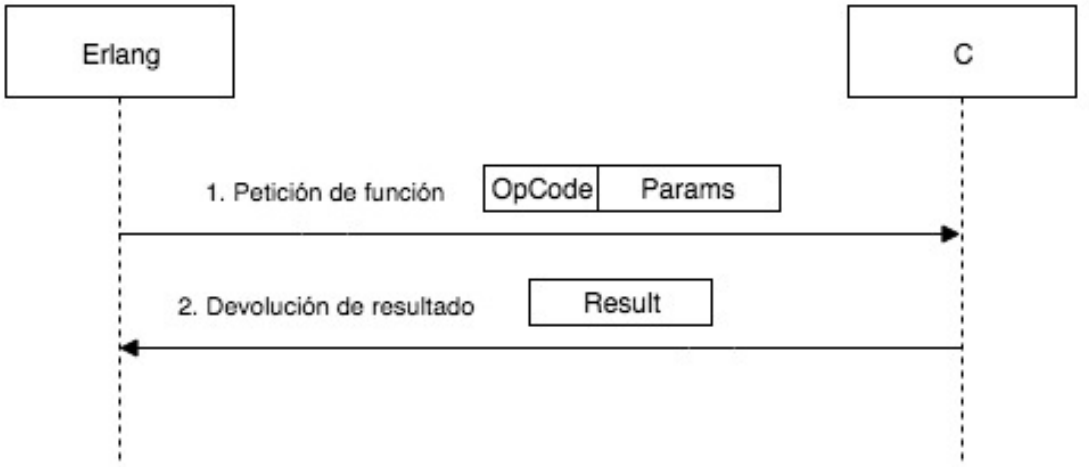

Figura 8.1: *Esquema de comunicación entre Erlang y C (primera versión)*

La inclusión de funciones de orden superior requiere un esquema de comunicación un poco más complejo, ya que las funciones que se pasan como parámetro solo pueden ser ejecutadas en la parte de Erlang. Esto implica que la parte de C deberá poder enviar también peticiones de función a Erlang para que las ejecute y le devuelva un mensaje de respuesta con el resultado cuando sea necesario. En resumen, ahora ambas partes pueden enviar y recibir ambos tipos de mensaje, lo cual eleva la complejidad a la hora de gestionar el bucle de comunicación.

Para que ambas partes puedan diferenciar en cada momento el tipo de mensaje que están recibiendo se ha incluido un nuevo código de 1 byte de longitud que precede a cada uno de ellos y cuya finalidad es diferenciar entre petición de función y devolución de resultados. Este código puede ser uno de los siguientes:

- `CALL_CODE`: Indica que el mensaje es una petición de ejecución de función.

- `RET_CODE`: Indica que el mensaje es una respuesta con el resultado de una operación.

Los siguientes listados muestran la definición de estas constantes tanto en Erlang como en C.

```
-define(CALL_CODE, 10).
-define(RET_CODE, 20).
```

```
#define CALL_CODE 10
#define RET_CODE 20
```

Dicho esto, vamos a tomar el ejemplo de la función `apply_int` presentada en la sección 8.1 para mostrar cómo queda el nuevo esquema de comunicación entre Erlang y C para funciones de orden superior (ver figura 8.2):

1. Desde Erlang, se a C una petición *CALL* para ejecutar la función `apply_int` con su código de operación y los parámetros.

2. Desde C, se comienza a ejecutar la función `apply_int`. Cuando esta necesite ejecutar la función que recibe como primer parámetro (`f1`), se envía una petición *CALL* a Erlang para que la ejecute. En este caso el código de operación toma el valor 1, ya que este es el primer parámetro de tipo funcional.

3. Desde Erlang, se ejecuta la función `f1` y se envía un mensaje de respuesta *RET* con el resultado de la ejecución.

4. Desde C, se continua la ejecución de `apply_int` y se envía a C otra petición *CALL* cuando se necesite ejecutar el segundo parámetro funcional (`f2`). En este caso el código de operación pasa a ser 2.

5. Desde Erlang, se ejecuta la segunda función (`f2`) y se devuelve el resultado mediante un mensaje de tipo *RET*.

6. Desde C, finaliza la ejecución de `apply_int` y se devuelve el resultado final a Erlang mediante un mensaje *RET*.

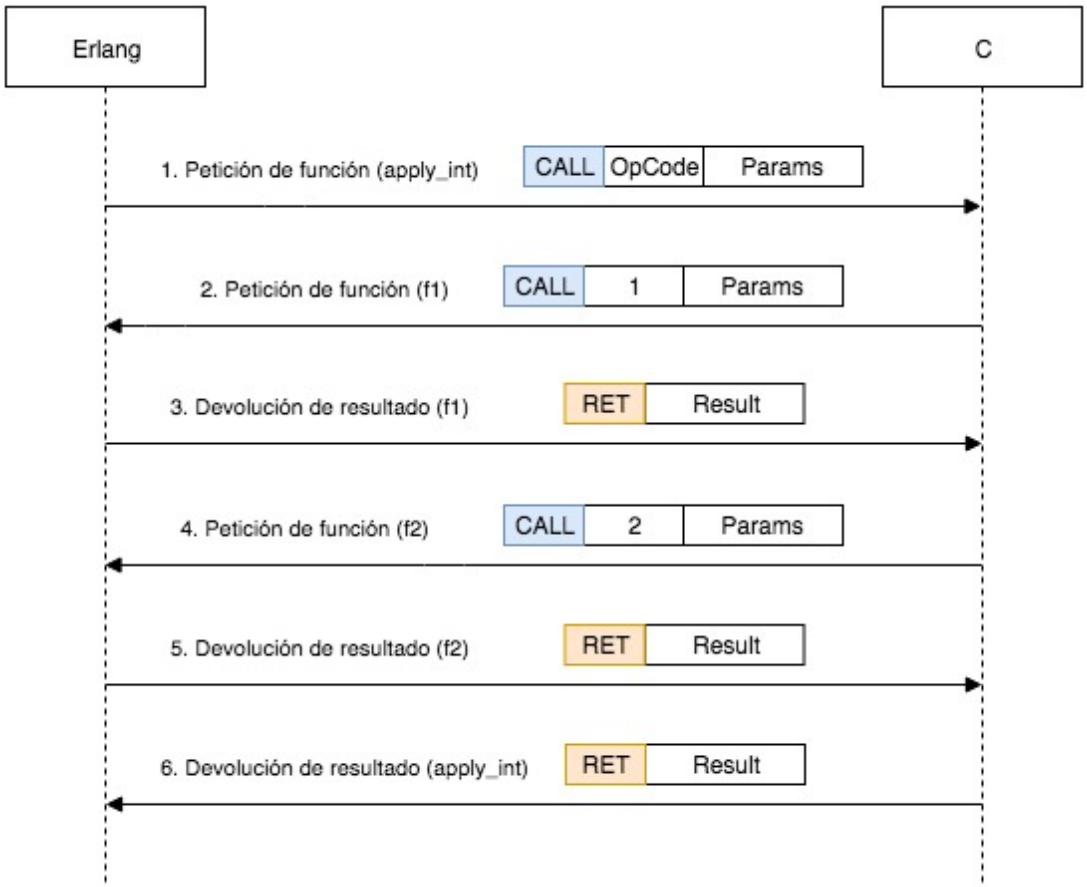

Figura 8.2: *Esquema de comunicación entre Erlang y C (segunda versión)*

Este esquema de comunicación solo se lleva a cabo cuando la función solicitada es de orden superior (tiene parámetros funcionales). En el resto de los casos se sigue el esquema simple visto en la figura 8.1, manteniendo en los mensajes la nueva cabecera con los códigos *CALL* y *RET*.

8.2.2. Actualización del bucle de procesamiento de mensajes de Erlang

Con la introducción de estos cambios, la función `loop_port_owner` que ejecuta el bucle de procesamiento de mensajes del *port owner* en Erlang debe ser actualizado para soportar este nuevo protocolo de comunicación. La principal dificultad viene debida al hecho de que

las funciones pasadas como parámetro a una función de orden superior podrían hacer uso de otras funciones también generadas por la herramienta. Esto involucra, en último término, el envío de un mensaje `CALL` por parte de Erlang mientras se está atendiendo una petición `CALL` proveniente de C.

De lo anterior se deduce que, en el código generado a partir del fichero de especificación, el bucle de recepción de mensajes del *port owner* ha de ser lo suficientemente flexible para poder soportar anidamiento de varias peticiones *CALL* consecutivas entre Erlang y C. Para hacer esto posible se han realizado las siguientes modificaciones:

- Se ha modificado el procesamiento de respuestas teniendo en cuenta los nuevos códigos *CALL* y *RET*.
- Se han generado dos funciones auxiliares denominadas `loop_receive_from_c` y `loop_receive_from_erlang` que se encargan de enviar/recibir los mensajes según la fase de comunicación en la que se encuentre.
- Se ha modificado la generación de la función `call_port_owner` que se encarga de enviar las peticiones al *port owner*. De modo que, si la petición contiene parámetros funcionales, se añade a la petición una lista con las funciones pasadas como parámetro. Si la petición no tiene parámetros funcionales, esta lista es vacía. Además, al principio del mensaje se añade el byte correspondiente al *CALL_CODE*.

```
call_port_owner(PortOwner, List, Funs) ->
   PortOwner ! { self(), { call, [?CALL_CODE | List], Funs }},
   receive
      {PortOwner, X} -> X
   end.
```

Con estos cambios, la secuencia de ejecución del bucle principal del *port owner* queda de la siguiente manera:

1. El *port owner* recibe desde una función Erlang una petición en `loop_port_owner`, la envía a través del puerto y llama a la función `loop_receive_from_c`.

2. La función `loop_receive_from_c` se encarga de esperar mensajes que provienen de la parte de código C. Si el mensaje recibido es de tipo *RET*, se limita a devolver el resultado al proceso `Pid` que lo había solicitado. En cambio, si el mensaje recibido es de tipo *CALL*, se crea un nuevo proceso auxiliar que se encarga de ejecutar la función identificada por `FunId` de la lista de funciones `Funs` (lista que contiene las funciones que han sido pasadas como parámetros de una función de orden superior) y enviar el resultado al *port owner*. Tras lanzar este proceso auxiliar llama a la función `loop_receive_from_erlang` indicándole la dirección del puerto y el identificador de este proceso recién creado (`PidN`).

3. La función `loop_receive_from_erlang` se encarga de recibir mensaje que provienen de la parte de código Erlang. Si el mensaje es de tipo `{result, Result}`, contiene el resultado (`Result`) de la función que estaba siendo ejecutada por el proceso auxiliar del paso anterior, que será enviado al programa C a través del puerto. En caso de recibir un mensaje de tipo `{call, List, Funs}`, se tratará de una nueva petición por parte de una función del programa Erlang. Esta petición se reenvía a través del puerto y se llama a la función `loop_receive_from_c` para seguir la ejecución de nuevo por el paso 2.

El siguiente listado muestra el código del *port owner* correspondiente al proceso que se acaba de describir:

```
loop_port_owner(Port) ->
   receive
      {Pid, {call, List, Funs}} ->
         % Se ha recibido una petición de llamada por parte del
         % proceso 'Pid'. Redirigimos esta petición al puerto.
         Port ! { self(), { command, List }},
         loop_receive_from_c(Port, Pid, Funs),
         loop_port_owner(Port)
   end.

loop_receive_from_c(Port, Pid, Funs) ->
   % Esperamos la respuesta a través del puerto
```

```
   receive
      % Si la respuesta es de tipo RET devolvemos el mensaje
      % al proceso 'Pid' que ha realizado la petición.
      { _, { data, [?RET_CODE | Msg] }} ->
         Pid ! {self(), {datalist, Msg}};
      % Si la respuesta es de tipo CALL...
      { _, { data, [?CALL_CODE, FunId | Msg] }} ->
         PortOwner = self(),
         % se ejecuta la función solicitada en un proceso aparte que
         % devuelve el resultado al port_owner
         PidN = spawn(fun() ->
            Fun = lists:nth(FunId, Funs),
            Result = Fun(Msg),
            PortOwner ! { self(), { result, [?RET_CODE | Result] } }
         end),
         % y se pasa el control al bucle loop_receive_from_erlang.
         loop_receive_from_erlang(Port, PidN),
         loop_receive_from_c(Port, Pid, Funs);
      Other ->
         Pid ! {self(), Other}
   end.

loop_receive_from_erlang(Port, PidCaller)   ->
   receive
      % Recibe una petición de llamada por parte del proceso 'Pid',
      % redirigimos esta petición al puerto y se devuelve el control
      % al bucle loop_receive_from_c
      { PidCaller, { call, List, Funs } } ->
         Port ! { self(), { command, List } },
         loop_receive_from_c(Port, PidCaller, Funs),
         loop_receive_from_erlang(Port, PidCaller);
      % Recibe el resultado de ejecución de un parámetro de función
      % y se lo envía a C a través del puerto.
      { PidCaller, { result, Result } } ->
         Port ! { self(), { command, Result }}
   end.
```

8.2.3. Actualización del bucle de procesamiento de mensajes de C

Por la parte del código C generado, también es necesario realizar cambios en el el bucle que procesa los mensajes para que tenga en cuenta las modificaciones realizadas.

Podemos destacar los siguientes cambios:

- Se ha separado de la función `main` el bucle principal de procesamiento de mensajes en una función `wait_for_input` independiente con el fin de que esta pueda ser llamada desde otras funciones y permitir un envío y recepción de mensajes más flexible de cara al nuevo esquema de comunicación. La función `wait_for_input` puede ser llamada en dos contextos distintos:
  - Desde la función `main`, con el fin de esperar la primera petición de llamada desde Erlang.
  - Tras enviar una petición *CALL* a Erlang, con el fin de esperar la respuesta, o bien esperar otra petición de llamada por parte de Erlang.
- Se ha añadido la comprobación del tipo de mensaje (*CALL* o *RET*). Ahora solo se ejecutará una función en caso de recibir un mensaje de tipo *CALL*, en cuyo caso la secuencia de ejecución es similar a la que había hasta el momento.
- Para los mensajes de tipo *RET* se provocará una salida de este bucle y será la propia función C que ha realizado la petición la que se encargue de obtener el resultado del buffer de entrada.

Introduciendo estos cambios el código queda de la siguiente manera:

```
void wait_for_input(byte *input_buffer, byte *output_buffer) {
   // Leemos el primer mensaje recibido
   int len_in = read_command(input_buffer);
   size_t len_out;
   byte *current_in = input_buffer;

   while (len_in > 0 && input_buffer[0] == CALL_CODE) {
      // Nos saltamos el byte del CALL_CODE en el buffer de entrada
      current_in++;

      // Obtenemos el código de operación
      int opcode;
```

```
      read_int(current_in, &opcode);

      // Llamamos a la función correspondiente
      handler current_handler = handlers[opcode];
      (*current_handler)(current_in, len_in, output_buffer, &len_out);

      // Escribimos por la salida estándar el resultado de output_buffer
      write_command(output_buffer, len_out);

      // Leemos el siguiente mensaje
      len_in = read_command(input_buffer);
      current_in = input_buffer;
   }

   // Si hemos salido del bucle puede ser por que se ha recibido un
   // carácter de fin de flujo, o bien porque se ha recibido un código
   // RET. En el primer caso finalizamos en el programa.
   if (len_in == 0) {
      printf("Closed.");
      exit(EXIT_SUCCESS);
   }
}

int main() {
   byte input_buffer[BUF_SIZE], output_buffer[BUF_SIZE];
   wait_for_input(input_buffer, output_buffer);
}
```

8.3. Cambios en la generación de código de funciones

Para finalizar este capítulo se van a presentar los últimos cambios que se han producido en la generación de código tras incluir el soporte para funciones de orden superior.

8.3.1. Cambios en el fichero de especificación

Comenzando por el fichero de especificación, es necesario establecer una nueva sintaxis para indicar los parámetros de función que son de tipo funcional. Para ellos se ha incluido el tipo `{function, ParamList, RetType}`, donde `ParamList` es una lista con los tipos de los

parámetros que recibe esta función y `RetType` es el tipo de dato que devuelve. Esta tupla se especifica dentro del campo del tipo de dato en la tupla `param_spec`.

El siguiente ejemplo muestra la especificación para la función `apply_int` definida en este capítulo:

```
{fun_spec, apply_int, "apply_int", [
      {param_spec, int, []},
      {param_spec, {function, [int], int}, []},
      {param_spec, {function, [int], int}, []}
   ],
   int, []}
```

Como puede verse en las líneas 3 y 4, esta función recibe dos parámetros funcionales y ambas reciben un entero como parámetro y devuelven otro entero.

8.3.2. Cambios en la generación de funciones en código Erlang

La generación de funciones en la parte de código Erlang también sufre modificaciones para soportar orden superior. La principal diferencia que encontramos entre una función de orden superior generada y el resto reside en la inclusión de una función envoltorio por cada una de las funciones que se pasan como parámetro.

Estas funciones envoltorio (o *fun wrappers*) se insertan justo a continuación de la serialización de parámetros y su objetivo es procesar la petición de la parte de C para ejecutar uno de los parámetros funcionales. Estas funciones se asignan a una variable (`FunN_Wrapper`) que se incluirá en la lista de funciones de la llamada al *port owner* mediante `call_port_owner`.

En el siguiente listado se muestra un ejemplo de función envoltorio que corresponde a una función que recibe como parámetro un entero y devuelve otro entero:

```
Fun1_Wrapper = fun(Buf) ->
   R0 = Buf,
   % Deserializar parámetro
   {P1, _R1} = parse_int(R0),
   % Llamar a función
   Result = Fun1(P1),
   % Serializar resultado
   int_to_bytelist(Result)
```

```
end,
```

Este tipo de funciones envoltorio tienen siempre tienen la misma estructura:

1. Recibe un *buffer* de bytes con los parámetros de la función que deben ser deserializados.

2. Se ejecuta la función a la que envuelve.

3. Se serializa el resultado y se devuelve como lista de bytes.

A modo de ejemplo de función de orden superior generada puede verse a continuación el código completo de la función `apply_int`:

```
apply_int(Int_1, Fun1, Fun2) ->
   % Serializamos el código de operación y los parámetros no funcionales
   Code = int_to_bytelist(771),
   Param1 = int_to_bytelist(Int_1),

   % Construimos un envoltorio de la función Fun1 que deserializa el
   % argumento y vuelve a serializar el resultado
   Fun1_Wrapper = fun(Buf) ->
      R0 = Buf,
      {P1, _R1} = parse_int(R0),
      Result = Fun1(P1),
      int_to_bytelist(Result)
   end,

   % Lo mismo para Fun2
   Fun2_Wrapper = fun(Buf) ->
      R0 = Buf,
      {P1, _R1} = parse_int(R0),
      Result = Fun2(P1),
      int_to_bytelist(Result)
   end,

   % Llamamos a call_port_owner con el código de operación, el primer
   % argumento, y las dos funciones envoltorio en el último argumento.
   ResultCall = call_port_owner(?PORT_NAME, [Code, Param1], [Fun1_Wrapper,
   Fun2_Wrapper]),
   case ResultCall of
      {datalist, DataList} ->
         {RetParamAux, _R1} = parse_int(DataList),
```

```
        RetParam1 = RetParamAux,
        RetParam1;
    Msg ->
        {error, Msg}
end.
```

8.3.3. Cambios en la generación de funciones en código C

Al igual que ocurre con la generación de código Erlang, las funciones de orden superior generadas presentan una estructura similar al resto con la diferencia de que además se generan funciones envoltorio para cada una de las funciones pasadas como parámetro.

En este caso las funciones envoltorio son independientes a la función principal generada y presentan una estructura un tanto mas compleja pero común a todas ellas. Tienen una signatura de tipos idéntica a la función envuelta.

Su funcionalidad se resume en los siguientes puntos:

1. Se escribe en el *buffer* de salida el byte del `CALL_CODE`, el byte del identificador de la función a ejecutar (no confundir con el código de operación que suele incluirse en las peticiones de llamada desde Erlang a C) y sus parámetros serializados.

2. Se envía el *buffer* al puerto y se llama a la función `wait_for_input` para esperar la respuesta o recibir nuevas peticiones en caso de que se produzcan.

3. Cuando se recibe la respuesta, continúa la ejecución, se deserializa el resultado y se devuelve.

El siguiente listado muestra el código una de las funciones envoltorio generadas para la función `apply_int`, que recibe y devuelve un entero.

```
int apply_int_wrapper_1 (int param1) {
   static byte input_buffer[BUF_SIZE];
   static byte output_buffer[BUF_SIZE];
   byte *current_out = output_buffer;
   size_t len_out = 0;
```

```
    // Código de llamada desde C a Erlang
    current_out = write_byte(CALL_CODE, current_out, &len_out);
    // Escribimos el identificador de la función a llamar (es importante
    // cuando se reciben varias funciones como parámetro, para así saber
    // a qué función llamar desde Erlang
    current_out = write_byte(1, current_out, &len_out);
    // Escribimos el argumento a pasar a la función.
    current_out = write_int(&param1, current_out, &len_out);
    write_command(output_buffer, len_out);

    // Esperamos la respuesta, o nuevas peticiones
    wait_for_input(input_buffer, output_buffer);

    byte *current_in = input_buffer;
    // Saltamos el byte de RET_CODE
    current_in++;
    int result;
    // Leemos el resultado.
    current_in = read_int(current_in, &result);

    // Y se lo devolvemos a la función apply_int
    return result;
}
```

Del mismo modo se generará otra función `apply_int_wrapper_2` que será la segunda función pasada a `apply_int`. La única diferencia con `apply_int_wrapper_1` es el identificador que se envía con la petición *CALL*. En este caso se enviará el número 2 como identificador, para indicar a Erlang que se llame a la segunda función pasada como parámetro, en lugar de la primera. La finalidad de estas funciones *wrapper* es ser pasadas como parámetro cuando el programa C ejecute la función solicitada por Erlang dentro de la función *handler*.

Una vez generadas las funciones envoltorio se genera la función *handler* que atiende la petición de Erlang, en este caso, haciendo uso de las funciones *wrapper*, quedando el código de la función `apply_int` de la siguiente manera:

```
void apply_int_Handler(byte *in, size_t len_in, byte *out, size_t *len_out) {
    byte *current_in = in, *current_out = out;
    *len_out = 0; current_in += 4;

    int var1;
```

```
    current_in = read_int(current_in, &var1);

    int retvar = apply_int(var1, apply_int_wrapper_1, apply_int_wrapper_2);
    current_out = write_byte(RET_CODE, current_out, len_out);
    current_out = write_int(&retvar, current_out, len_out);
}
```

Con esto último finaliza la descripción de los cambios realizados en la generación de código para el soporte de funciones de orden superior. Este capítulo supone el último en cuanto a aspectos de generación de código, de modo que en el siguiente se abordarán las conclusiones obtenidas tras la realización de este trabajo, así como aspectos de trabajo futuro.

Capítulo 9

Conclusiones

Llegados a este punto solo queda hacer balance del trabajo realizado, remarcar los resultados conseguidos, así como los problemas encontrados, y establecer los siguientes pasos en el desarrollo de este proyecto de cara al trabajo futuro.

9.1. Resultados conseguidos

Como se comentaba en el capítulo 1, el objetivo principal de este trabajo es el desarrollo de una herramienta de generación automática de *bindings* para el uso de librerías de C en programas Erlang que abarque los distintos aspectos que han ido tratándose en cada uno de los capítulos de esta memoria, así como aplicar dicha herramienta a la librería SDL para desarrollar en Erlang un sistema de prueba que demuestre la validez del código generado.

Finalmente la herramienta desarrollada cumple con los requisitos establecidos en la planificación del trabajo y, valiéndose del mecanismo de puertos para la interoperabilidad entre Erlang y C, el código generado presenta las siguientes funcionalidades:

- Tratamiento de tipos básicos.
- Tratamiento de macros y constantes.
- Adaptaciones de las funciones propias de la librería C (los *bindings* propiamente dichos).

- Tratamiento de enumerados, estructuras y uniones.
- Tratamiento de punteros de C.
- Tratamiento de *arrays* de C.
- Gestión automática de memoria dinámica para punteros de C (recolección de basura).
- Tratamiento de funciones de orden superior.

Tras las pruebas realizadas con el sistema desarrollado con la librería SDL, podemos observar que el código generado es plenamente funcional y permite el desarrollo en Erlang de programas completos sin limitaciones reseñables. Haciendo uso de este código generado, se ha desarrollado enteramente en Erlang una réplica del programa de prueba inicial desarrollado en C (sección 2.7), obteniendo unos resultados similares en ambas versiones.

Sin duda, el trabajo realizado abre la puerta la utilización de esta herramienta con otro tipo de librerías de C, como pueden ser *Z3*, *GTK+*, *GObject* o *GLib*. La herramienta generadora de código ha sido implementada pensando en su utilización con cualquier librería de C, por lo que puede resultar de gran utilidad como apoyo al desarrollo de programas en Erlang, aportando funcionalidades de las que no dispone el propio lenguaje ni las librerías de terceros.

9.2. Problemas encontrados

A lo largo del desarrollo de este trabajo se han encontrado una serie de problemas, de entre los cuales podemos destacar los siguientes:

- Problemas en la utilización de NIFs como mecanismo de interoperabilidad entre Erlang y C a la hora de recibir eventos de SDL. Como ya se comenta en la sección 2.8, esta funcionalidad debe ser ejecutada por el hilo principal del programa, cosa que no siempre ocurre debido a la propia gestión de hilos que hace Erlang al llamar a las

NIFs. La solución a este problema ha sido hacer uso de puertos como mecanismo de interoperabilidad.

- Problemas a la hora de tratar *unions* en Erlang como registros con sus elementos y no solo a través de punteros. La imposibilidad de saber en tiempo de ejecución el tipo de dato que contiene el *union* dificulta su conversión en un registro. De momento no se ha implementado una solución a este problema.

- Problemas en el bucle de gestión de mensajes del *port owner* a la hora de gestionar las peticiones para funciones de orden superior. Este problema hacía que las peticiones, por parte de C, de ejecución de funciones pasadas como parámetro no fueran gestionadas de forma correcta, impidiendo recibir otras peticiones procedentes de Erlang de antes de su finalización. Como solución se ha implementado un bucle más complejo que separa en funciones diferentes las peticiones de Erlang y las de C y, además, se crea un proceso independiente para la ejecución de parámetros funcionales para evitar errores en la recepción de mensajes del *port owner*.

9.3. Trabajo futuro

Para finalizar, se han establecido los siguientes puntos a tener en cuenta de cara a un trabajo futuro:

- Uso de *Erl_Interface* para la serialización de datos en lugar de realizarla manualmente como se viene haciendo hasta el momento en el código generado. La librería *Erl_Interface* [1] ofrece funciones para el tratamiento de tipos de datos entre C y Erlang, serialización, deserialización, etc.

- Mejorar el tratamiento de los tipos *union* con el fin de poder manejarlos en Erlang como registros con sus elementos y no solo como punteros. Muchos de los tipos *union* que se utilizan en la práctica contienen un atributo discriminante (habitualmente un número

entero), que indica cuál de los campos de la estructura es el asignado actualmente. Es posible extender nuestro fichero de especificación con el fin de señalar, dentro de un tipo *union*, el campo discriminante y la correspondencia entre los posibles valores del campo discriminante y los demás campos del tipo.

- Añadir soporte para funciones de orden superior que devuelvan otras funciones. Como se menciona en el capítulo 8, de momento solo es posible generar *bindings* de funciones que reciben otras funciones como parámetros, siempre y cuando estas no sean a su vez de orden superior. Esto puede abordarse incluyendo nuevas funciones auxiliares en el código C que permitan, a partir de un puntero a función, llamar a la función apuntada, utilizando los mecanismos de serialización/deserialización ya conocidos para transformar los parámetros y el resultado.

- Hacer que el código sea generado de manera modular. En la versión actual de la herramienta tanto el código C como el código Erlang se generan en un único fichero, salvo algunos elementos definidos en un fichero de cabecera (`.hrl`). En este caso, el programador podría declarar distintas secciones en el fichero de especificación, generando la herramienta un par de ficheros distintos (Erlang/C) para cada una de ellas.

- Asegurar que la herramienta sea multiplataforma. Como se menciona en el capítulo 2, la herramienta ha sido probada en entornos con *macOS 10.13.6* y *Linux (Fedora 28)* obteniendo resultados satisfactorios. No obstante, la herramienta asume que los punteros con los que trabaja tienen siempre un tamaño de 8 bytes, lo cual puede causar incompatibilidades en sistemas en los que esta suposición no sea cierta. Estos aspectos dependientes de la plataforma pueden solventarse mediante un mecanismo de configuración automática (similar al sistema *GNU Automake* [11]), o bien mediante la configuración manual, introduciendo las directivas adecuadas en el fichero de especificación.

Chapter 9

Conclusions

At this point it is only necessary to evaluate the work done, highlight the results achieved, as well as the problems encountered, and sketch the following steps in the development of this project as future work.

9.1. Results achieved

As discussed in the chapter 1, the main objective of this work is the development of a tool for the automatic generation of bindings that allows one to use C libraries in Erlang programs, and that covers the different aspects that have been treated in each of the chapters of this document. We have also applied this tool to the SDL library to develop in Erlang a proof-of-concept that demonstrates the validity of the generated code.

Finally, the tool developed meets the requirements established in the work planning and, using the ports mechanism for interoperability between Erlang and C, the generated code has the following functionalities:

- Management of basic types.
- Management of macros and constants.
- Adaptations of the functions of the C library (the bindings themselves).
- Management of enumerations, structures and unions.

- Management of C pointers.
- Management of C arrays.
- Automatic management of dynamic memory for C pointers (garbage collection).
- Management of higher order functions.

After the tests carried out with the system developed with the SDL library, we can see that the generated code is fully functional and allows the development in Erlang of complete programs without notable limitations. Using this generated code, a replica of the initial test program developed in C (section 2.7) has been developed entirely in Erlang, obtaining similar results in both versions.

Undoubtedly, the work done allows one to use of this tool with other kind of C libraries, such as *Z3*, *GTK+*, *GObject* or *GLib*. The code generation tool has been implemented in order to be used with any C library, so it can be very useful to support the development of Erlang programs, by providing functionality covered neither by the language itself nor by third-party libraries.

9.2. Problems encountered

Throughout the development of this work have been a number of problems, among which we can highlight the following:

- Problems in the use of NIFs as an interoperability mechanism between Erlang and C when receiving SDL events. As already mentioned in the section 2.8, this functionality must be executed by the main thread of the program, something that does not always happen due to the thread management that Erlang does when calling the NIFs. The solution to this problem has been to make use of ports as an interoperability mechanism.

- Problems when dealing with unions in Erlang as records with their elements and not only through pointers. The impossibility of knowing, at runtime, the type of data that contains the union hinders its conversion into a registry. A solution to this problem has not been implemented yet.

- Problems in the message management loop of the *port owner* when managing requests for higher order functions. This problem caused that the requests, from the C part, of execution of functions passed as a parameter were not managed correctly, preventing receiving other requests from Erlang before its completion. As a solution, a more complex loop has been implemented. This loop separates the Erlang and C requests into different functions and, in addition, creates an independent process for the execution of functional parameters to avoid errors in receiving messages from *port owner*.

9.3. Future work

Finally, the following points have been taken into account for future work:

- Use of *Erl_Interface* for the serialization of data instead of doing it manually as it has been done up to now in the generated code. The *Erl_Interface* [1] library offers functions for the processing of data types between C and Erlang, serialization, deserialization, etc.

- Improve the treatment of the union types in order to be able to handle them in Erlang as records with their elements and not just as pointers. Many of the union types that are used in practice contain a discriminant attribute (usually an integer), which indicates which of the fields of the structure is currently assigned. It is possible to extend our specification file in order to indicate, within an union type, the discriminant field and the correspondence between the possible values of the discriminant field and the other fields of the type.

- Add support for higher order functions that return other functions. As mentioned in the chapter 8, for the moment it is only possible to generate bindings of functions that receive other functions as parameters, as long as they are not of a higher order. This can be addressed by including new auxiliary functions in the C code that allow, from a pointer to function, call the function pointed, using the known serialization/deserialization mechanisms to transform the parameters and the result.

- Make the code generated more modular. In the current version of the tool, both the C code and the Erlang code are generated in a single file, except for some elements defined in a header file (`.hrl`). In this case, the programmer could declare different sections in the specification file, generating the tool a couple of different files (Erlang/C) for each of them.

- Ensure that the tool is cross-platform. As mentioned in the chapter 2, the tool has been tested in *macOS 10.13.6* and *Linux (Fedora 28)* environments obtaining satisfactory results. However, the tool assumes that the pointers with which it works always have a size of 8 bytes, which can cause incompatibilities in systems where this assumption is not true. These aspects depending on the platform can be solved by means of an automatic configuration mechanism (similar to the system *GNU Automake* [11]), or by manual configuration, introducing the appropriate directives in the specification file.

Bibliografía

[1] Ericsson AB. Erlang/OTP system documentation 10.0, 19 junio 2018.

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang, 1993.

[3] Marco A Peña Basurto and José M Cela Espín. *Introducción a la programación en C*, volume 42. Univ. Politèc. de Catalunya, 2000.

[4] David M. Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[5] Andrew Bennet. Latency of native functions for Erlang and Elixir. Potatosalad blog. Disponible en https://potatosalad.io/2017/08/05/latency-of-native-functions-for-erlang-and-elixir, agosto 2017. Online; accedido 2 ago 2018.

[6] Stephen Cass. The 2018 top programming languages. https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages. Acceso: 2018-08-31.

[7] PARAPLUU (Programming Language group. Division of Computing Science. Uppsala University). Erlang NIF wrapper generator. https://github.com/parapluu/nifty. Acceso: 2018-08-31.

[8] Andreas Jaeger. Porting to 64-bit GNU/Linux systems. *Proceedings of the First Annual GCC Developers' Summit*, pages 107–120, 2003.

[9] Maxim Kupriianov. Automatic C-Go Bindings Generator for Go Programming Language. https://github.com/xlab/c-for-go. Acceso: 2018-08-31.

[10] Andreas Löscher and Konstantinos Sagonas. The nifty way to call hell from heaven. In *Proceedings of the 15th International Workshop on Erlang*, Erlang 2016, pages 1–11, New York, NY, USA, 2016. ACM.

[11] David MacKenzie, Tom Tromey, Alexandre Duret-Lutz, Ralf Wildenhues, and Stefano Lattarini. *GNU Automake manual*. Free Software Foundation, 2018. Disponible en https://www.gnu.org/software/automake/manual/.

[12] Loïc Hoguin (Nine Nines). SDL2 Erlang NIF. https://github.com/ninenines/esdl2. Acceso: 2018-08-31.

[13] Michal Ptaszek. Python bindings in erlang using NIFs. https://github.com/paulgray/Pytherl. Acceso: 2018-08-31.

[14] Aur Saraf. Erlang SDL bindings. https://github.com/SonOfLilit/esdl. Acceso: 2018-08-31.

[15] Víctor Theoktisto. Enfoque funcional robusto con aspectos formales en la enseñanza de lenguaje C. pages 159–160, 2016.